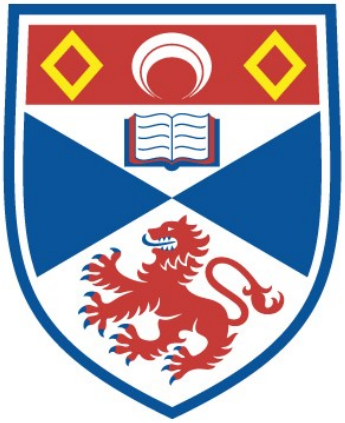
MODELLING RECOVERY IN DATABASE SYSTEMS

Stephan J. G. Scheuerl

A Thesis Submitted for the Degree of PhD
at the
University of St Andrews



1998

Modelling Recovery in Database Systems

A thesis submitted to the

UNIVERSITY OF ST ANDREWS

for the degree of

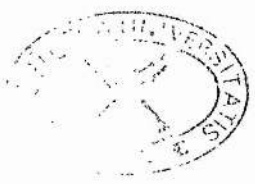
DOCTOR OF PHILOSOPHY

By

Stephan J.G. Scheuerl

School of Mathematical and Computational Sciences
University of St Andrews

August 1997



ProQuest LLC.
789 East Eisenhower Parkway
P.O. Box 1346
Ann Arbor, MI 48106 – 1346

(i) I, Stephan J.G. Scheuerl, hereby certify that this thesis, which is approximately 45000 words in length, has been written by me, that it is the record of work carried out by me and that it has not been submitted in any previous application for a higher degree.

date ..... signature of candidate .....

(ii) I was admitted as a research student in October 1993 and as a candidate for the degree of Doctor of Philosophy in October 1994; the higher study for which this is a record was carried out in the University of St Andrews between 1993 and 1997.

date ..... signature of candidate .....

(iii) I hereby certify that the candidate has fulfilled the conditions of the Resolution and Regulations appropriate for the degree of Doctor of Philosophy in the University of St Andrews and that the candidate is qualified to submit this thesis in application for that degree.

date ..... signature of supervisor .....

In submitting this thesis to the University of St Andrews I understand that I am giving permission for it to be made available for use in accordance with the regulations of the University Library for the time being in force, subject to any copyright vested in the work not being affected thereby. I also understand that the title and abstract will be published, and that a copy of the work may be made and supplied to any bona fide library or research worker.

date ..... signature of candidate .....

Acknowledgements

I would like to thank my supervisor Ron Morrison for all the advice and support he has offered towards this research.

Dave Munro must also be thanked for his enthusiastic discussions on all aspects of this work.

Thanks to Eliot Moss for his suggestions and contributions that have lead to the work presented. Equally I thank Richard Connor for his contributions and encouragement.

Thanks also go to Graham Kirby, Malcolm Atkinson, Robin Stanton, Fred Brown, Steve Blackburn, and Dave Hulse for their advice and motivation.

Finally to Duncan, Dominic and Dharini for their alternative views on my research and to Shona for her encouragement when there seemed no end in sight.

Abstract

The execution of modern database applications requires the co-ordination of a number of components such as: the application itself, the DBMS, the operating system, the network and the platform. The interaction of these components makes understanding the overall behaviour of the application a complex task. As a result the effectiveness of optimisations are often difficult to predict. Three techniques commonly available to analyse system behaviour are empirical measurement, simulation-based analysis and analytical modelling.

The ideal technique is one that provides accurate results at low cost. This thesis investigates the hypothesis that analytical modelling can be used to study the behaviour of DBMSs with sufficient accuracy. In particular the work focuses on a new model for costing recovery mechanisms called MaStA and determines if the model can be used effectively to guide the selection of mechanisms.

To verify the effectiveness of the model a validation framework is developed. Database workloads are executed on the flexible Flask architecture on different platforms. Flask is designed to minimise the dependencies between DBMS components and is used in the framework to allow the same workloads to be executed on a various recovery mechanisms. Empirical analysis of executing the workloads is used to validate the assumptions about CPU, I/O and workload that underly MaStA. Once validated, the utility of the model is illustrated by using it to select the mechanisms that provide optimum performance for given database applications.

By showing that analytical modelling can be used in the selection of recovery mechanisms, the work presented makes a contribution towards a database architecture in which the implementation of all components may be selected to provide optimum performance.

Contents

1 Introduction

The work presented makes a contribution towards realising a flexible database architecture that may be configured to obtain the optimum performance for any particular application. To optimise such a system effectively a technique is required that allows the behaviours of different system configurations to be compared. This thesis investigates the hypothesis that analytical modelling of the database system and the application may be employed to make accurate comparisons. In particular the work develops and validates a new cost model, called MaStA, to show that analytical techniques can be used to guide the choice of recovery mechanisms for optimum performance.

1.1 Components of DBMSs

To aid in the design and implementation of database management systems (DBMSs), the major tasks dealt with by these systems can be logically partitioned into a number of components as illustrated in Figure 1.1.

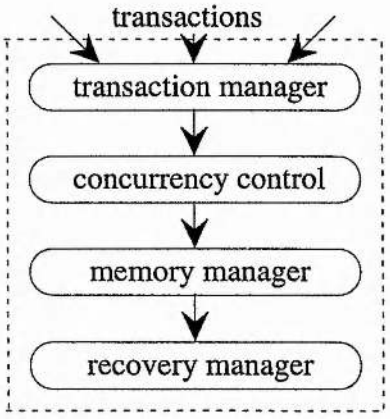


Figure 1.1: Logical Components of a DBMS

- Transactions access the database through a transaction manager. The transaction manager receives operations from transactions and forwards them to other components of the DBMS.

- Concurrency control is responsible for the correct concurrent execution of transactions. This is achieved by controlling the execution of operations on the database in such a manner that ensures transactions adhere to the constraints of the particular concurrency model employed.

- Memory management traditionally deals with caching the database in main memory. Recently, memory management has also taken on the role of

controlling the movement of data between main memory and high speed caches, and clustering strategies within the database.

- Recovery management is responsible for ensuring that the database is fault tolerant - the data is not corrupted even in the event of software, system or media failure.

Although there are many documented mechanisms for each of these components the convention is that only one implementation of each component is embedded into a particular database system. Furthermore, implementations of DBMSs often deviate from the logical partitioning of the systems (Figure 1.1) by combining the implementations of various components. One justification for such vertical structuring may be the perceived performance gains obtained over layered implementations. On the other hand integrating implementations of DBMS components may introduce dependencies between the layers that may in turn increase the complexity of altering the implementation of any single component. The next section proposes a flexible architecture that reflects the logical view of DBMSs in a layered implementation.

1.2 Configuring DBMSs

In many conventional DBMSs the particular styles of memory management, recovery mechanism and concurrency model employed are designed to provide good average performance for applications executed on the systems. However it is no longer clear if the analysis on which the designs of these systems are based is still valid. Current trends in application styles, hardware configurations and operating systems weaken many of the assumptions made by early studies. Furthermore recent research [WJN+95, SCM+95a] suggests that the models of computation, memory management, CPU and I/O have been too simple, thus casting doubt on their accuracy. Due to this, it may be argued that a new approach to maximising the performance of DBMSs must be taken - one that takes into account the current state of technology and considers application style and workload in the configuration of each component of the system.

The aim of this work is to provide a more flexible approach to maximising the performance of a particular database application. The approach is unconventional by taking the form of a flexible database architecture (Figure 1.2) that is configured according to the application workload. The logical components are separated in the proposed architecture to ensure that the implementation of each component is independent of any other. This approach provides the flexibility required to make

changes to individual components and provides the potential to optimise performance for an application.

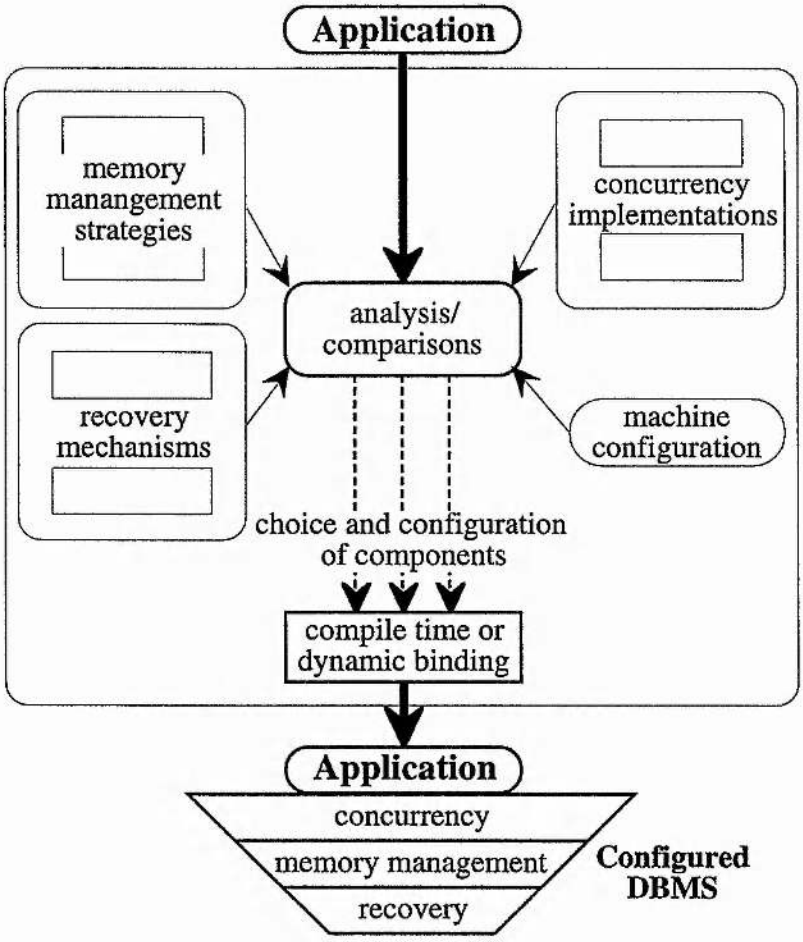


Figure 1.2: Conceptual View of the Flexible DBMS Architecture

For each component, analysis is performed to determine the elements of the workload that contribute to the costs incurred. Workload properties relevant to memory management for example include the volume and locality of data read and updated by the application. The analysis also takes into account the particular machine configuration such as the size of main memory and the characteristics of the disks available. The costs incurred by the application on various implementations of a component are compared and the implementation with the lowest cost is selected. This process is repeated for each DBMS component resulting in a configuration that provides optimum performance for the particular application. Once a configuration is obtained the components are bound with the application. Binding may be performed at compile time to take advantage of compile time optimisations or bound dynamically providing opportunities to configure the system at run time.

To make policy decisions regarding how each component should be configured, a technique is required that allows the different implementations of each component to be compared. Three commonly used techniques are available. These are analytical, simulation and empirical based analysis:

- Empirical measurement involves running applications or benchmarks on implemented DBMSs taking measurements using hardware or software monitoring.

- Simulation based analysis comprises of a number of programs that capture the characteristics of a component. By running these programs the behaviour of the component is approximated and so its performance may be studied. The simulations are based on a number of assumptions to reduce the complexity of each component.

- Analytical modelling allows the performance characteristics of each DBMS component to be derived mathematically. This involves the construction of a number of parameterised equations that approximate the attributes of the components in terms of workload characteristics. As with simulations a number of simplifying assumptions are made about behaviour.

In analytical modelling and simulations a number of assumptions are required to make the analysis tractable [Leu88]. These assumptions must be sufficiently understandable to allow the analytical model or simulations to be constructed. Generally simulations permit more details of a system to be incorporated - details that are often difficult to include in analytical models. A drawback of simulation models is that they tend to be more expensive in terms of programming, debugging and validation to develop, and more expensive to use than analytical models. The fine grained analysis that can be performed using empirical measurement, the most expensive form of analysis, ensures that the results obtained are normally the most accurate of the three techniques available. Measurement of systems using empirical analysis is often performed using synthetic application workloads produced by benchmark suites such as OO1 [CS92] and OO7 [CDN93].

1.3 Contribution

Realising an architecture in which all DBMS components are engineered to suit the application is a large and complex task. Hence this thesis focuses on developing an analytical cost model called MaStA [SCM+95a] for recovery mechanisms, and shows that the model can be used to guide the configuration of the recovery component of a flexible architecture. MaStA analyses the workload of the application to determine the

number of I/O operations incurred by each recovery mechanism available and analyses the platform to ensure that cost predictions are platform specific. Comparisons of the resulting cost predictions can then be used to select the recovery mechanism that incurs the lowest cost.

To promote confidence in the MaStA model a validation framework is developed. The framework involves running workloads typical of database applications on various recovery mechanisms on different platforms. Empirical analysis of the executing workloads is used to validate assumptions about CPU, I/O and workload that underly MaStA. The flexible recovery management required in the framework to execute workloads on various mechanisms is provided by the Flask architecture [MCM+94]. Flask goes some way to realising the flexible architecture proposed. An attraction of Flask is that the responsibility of recovery management and concurrency control are separated thereby enabling the implementations of recovery schemes to be developed and altered independently of concurrency control. The flexibility in the recovery component of Flask is achieved through an interface that places few constraints on the mechanism used.

A wide variety of recovery schemes have been documented [AS82, Grá78, Lor77, RO91] any of which may be used in architecture. In the original instantiation of Flask, concurrent shadow paging [Mun93] is employed. This thesis develops two other schemes: a log-structured mechanism and a log-based mechanism called DataSafe [SCM+96]. By providing a choice, the work provides a means to execute the same database workloads over different recovery mechanisms in the framework used to validate the MaStA cost model. Furthermore, incorporating different recovery mechanisms provides opportunities to perform empirical analysis on Flask to illustrate the necessity for the proposed flexible architecture. Once MaStA is validated the utility of the model is illustrated by using it guide the choice of recovery mechanism in Flask to provide optimum performance for given database applications

1.4 Thesis Structure

The need for recovery mechanisms in database systems is introduced in Chapter 2, followed by a description of a classification used to distinguish between mechanisms. A discussion of commonly used recovery schemes is accompanied by a summary of existing analytical and empirical studies of recovery mechanisms. Benchmarks frequently used in empirical studies of DBMS are reviewed. A summary of different concurrency models is included along with a description of the Flask architecture to provide an insight into how the logical concurrency and recovery components of a DBMS may be separated.

Chapter 3 provides an overview of the flexible recovery manager used in Flask and includes a description of two mechanisms developed to provide alternatives to the scheme used in the first instantiation of Flask.

The new analytical model used to select the appropriate recovery mechanism for a particular application is developed in Chapter 4. The assumptions of the model and a validation framework are discussed in Chapter 5 and a description of the empirical measurements and simulation experiments performed to validate the assumptions are provided in Chapter 6. A worked example given in Chapter 7 illustrates how MaStA can be used in Flask to select the mechanism that incurs the lowest cost for given database applications.

2 Background

2.1 Introduction

The Flask architecture provides the potential to engineer recovery management independently from other DBMS components in order to obtain the optimum performance for a particular application. Early attempts to increase the performance of database systems have resulted in numerous designs for recovery mechanisms. This chapter introduces the requirement for these schemes and discusses the trade-offs between various recovery mechanism designs, any of which may be adopted in the Flask architecture.

Previous analysis of DBMSs develop analytical models to compare the systems mathematically or use benchmarks to provide workloads for empirical measurement. These studies along with summaries of commonly used benchmarks are discussed to provide the background for a new analytical model used to drive the Flask architecture.

2.2 Recovery Management

2.2.1 Introduction

Traditionally, recovery management is tightly coupled to the implementation of transactions in DBMSs. Transactions [Dav73, EGL+76, Dav78] were introduced into database systems to allow activities to execute concurrently, thus increasing database resource utilisation. Each transaction is a unit of work consisting of reads and possibly updates to a database. A transaction completes by either committing or aborting as a unit. When a transaction commits, all updates performed by the transaction are made permanent in the database and visible to other transactions. In contrast, when a transaction aborts, all updates are discarded and the database is left in a state it would have been in if the transaction had never executed. This is known as the atomicity or all-or-nothing property of transactions - either all or none of a transaction's updates are reflected in the database. Durability is the property that a successfully committed transaction's updates survive failures. Recovery management is the DBMS component responsible for providing the durability properties of transactions in the presence of failures. The three types of failure that the recovery manager must deal with are media failure, system failure and transaction abort.

- media failure: These failures occur from the breakdown of hardware and potentially causes the loss of data on both volatile and non-volatile storage. In

such an event the data may be restored from a mirrored disk [BT85] or from an archived version.

- system failure: These failures occur due to the loss of data from volatile storage only and potentially cause inconsistencies in the *materialised database* [HR83]. The term materialised database is used to describe the state of the database only, i.e. taking no account of additional data that may be recorded during normal processing to recover the database to a consistent state. The recovery manager ensures that on restart all updates made by committed transactions are durable and that all updates of non-committed transactions are removed. Since system failures may also occur during restart the recovery process must be idempotent. That is, restart may begin and fail a number of times, eventually succeeding, resulting in the same state as if the initial restart had succeeded.

- transaction abort: A transaction is said to abort if it is terminated before it commits. All updates made by the transaction to the materialised database must be removed by the recovery manager. This is known as transaction *rollback*.

This thesis concentrates on the provision for recovery after system failure and transaction abort. Recovery from media failure requires additional mechanisms such as disk mirroring [BT85, SO91, OS93] or RAID [PGK88] which are beyond the scope of this thesis, but may be included in future work as a separate DBMS component in the flexible architecture outlined in Figure 1.1.

Figure 2.1 illustrates the principle behind all recovery mechanisms designed to deal with soft failure. The database is held on non-volatile storage such as disk. Read operations cause data to be faulted into a cache held in main memory where the data may be updated. During a commit, or in some cases during the transaction, the updated data is transferred back to non-volatile storage.

The non-volatile storage is partitioned into two logical areas: the database itself and an extra partition traditionally known as the log to record the information necessary for recovery. In some cases, the database and the log are two distinct areas of non-volatile storage, such as a database file, and a log [Gra78] or difference [AS82] file. In others, such as shadow paging [Lor77] or log-structuring [RO91] the database and the log are intermingled on a single area of storage. In each case the recovery information is maintained in the log so that inconsistencies may be removed from the materialised database on restart.

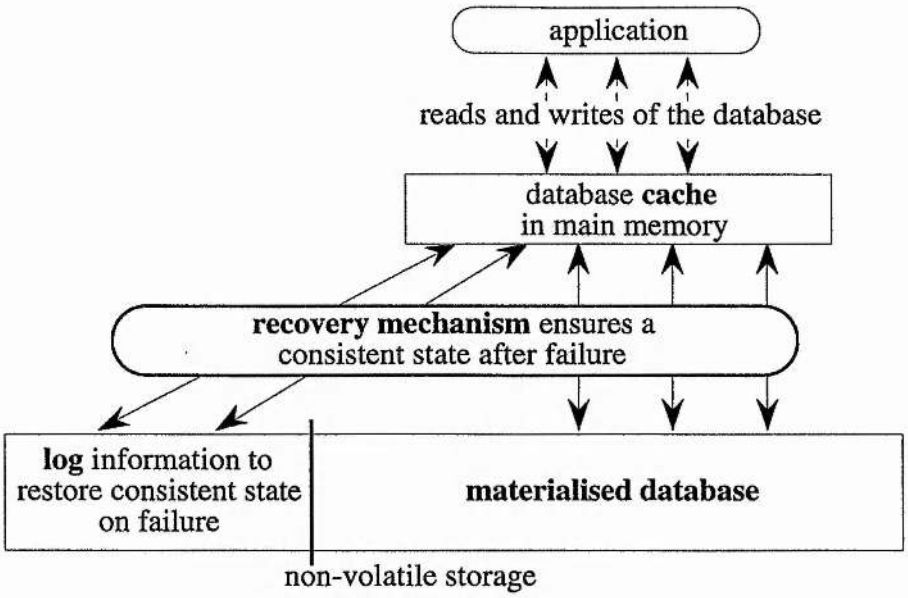


Figure 2.1: The General Structure of a Recovery Mechanism

2.2.2 Classification of Recovery Mechanisms

To understand the trade-offs between recovery mechanisms and to provide a technique for distinguishing between these schemes it may be helpful to develop classifications of the properties of recovery mechanisms. Haerder and Reuter [HR83] stratify recovery into a hierarchy of propagation, page replacement, end-of-transaction processing and checkpointing strategies adopted by page-based recovery mechanisms in transactional database systems. Haerder and Reuter's classification subsumes another classification, sometimes referred to as the *undo/redo* categorisation [BGH83, HR83]. The *undo/redo* scheme describes mechanisms in terms of the operations performed on restart to bring the materialised database to a consistent state after system failure.

Propagation is the process of making committed updates visible in the materialised database. The propagation strategy of a mechanism is *atomic* if a transaction's updates to the database are performed as a unit when the transaction commits. In other words either all or none of a committing transaction's updates become part of the database. Such schemes are often called *no-undo/no-redo* since the database is always left in a consistent state after a system failure and hence require no recovery operations on restart. The propagation strategy is ¬*atomic* if commit propagation to the database is interruptable by system failures. If a system crash occurs during ¬*atomic* propagation, the materialised database may be left in an inconsistent state after a crash.

A recovery mechanism's page replacement strategy is *steal* if cache pages updated by a transaction may be written in place to the database before the transaction commit completes. Mechanisms exhibiting *steal* strategies require that information is recorded in the log to remove non-committed updates during transaction rollback or if a system failure occurs before the transaction successfully commits. Such mechanisms may be classified as requiring *undo* operations on restart after system failures. A mechanism is ¬*steal* if the pages updated by a transaction are held in main memory or in the log until after the transaction commits. The materialised database therefore never contains non-committed updates and so no *undo* operations are required after a crash.

A mechanism is *force* if updated pages are propagated to the database during a transaction commit and ¬*force* if propagation is deferred until after commit time. A ¬*force* strategy must write updated pages to the log to ensure propagation can be performed at a later time, and hence may be classified as *redo*. Mechanisms exhibiting *force* end-of-transaction processing are *no-redo* since all committed updates are present in the materialised database after a system failure.

In *redo* recovery mechanisms the amount of recovery information required in the log is conceptually unbounded. Checkpointing schemes are used to limit the amount of this information. A checkpoint involves writing to the database, updates held in the log and writing a checkpoint record to the log to indicate the fact. The checkpointing strategy adopted by a recovery mechanism determines the frequency of checkpoints and the amount of work performed during each checkpoint:

- Transaction-oriented checkpoints (TOC): these occur each time a transaction commits and are associated with a *force* propagation strategy.

- Transaction-consistent checkpoints (TCC): in-progress update transactions are allowed to terminate and new update transactions are blocked. All updates are then propagated to the database after which normal execution is resumed.

- Action-consistent checkpoints (ACC): These are generated in a similar manner to transaction-consistent checkpoints but at an operational level instead of at the level of transactions. All update operations are finished and new update operations are blocked until after the checkpoint completes. Changes are then propagated to the database.

- Fuzzy checkpointing: these checkpointing schemes reduce the amount of propagation that takes place at checkpoint time. Instead of propagating all updated pages on every checkpoint, only a fraction of the pages that have not

been propagated since the last checkpoint are propagated to the database. The number of pages propagated and the nature of the checkpoint trigger are determined by the particular fuzzy checkpointing scheme employed.

The classification depicted in Figure 2.2, taken from [HR83], stresses the possible combinations of strategies that may be used by recovery mechanisms. The fact that there are numerous strategies makes the comparison of recovery schemes a complex task.

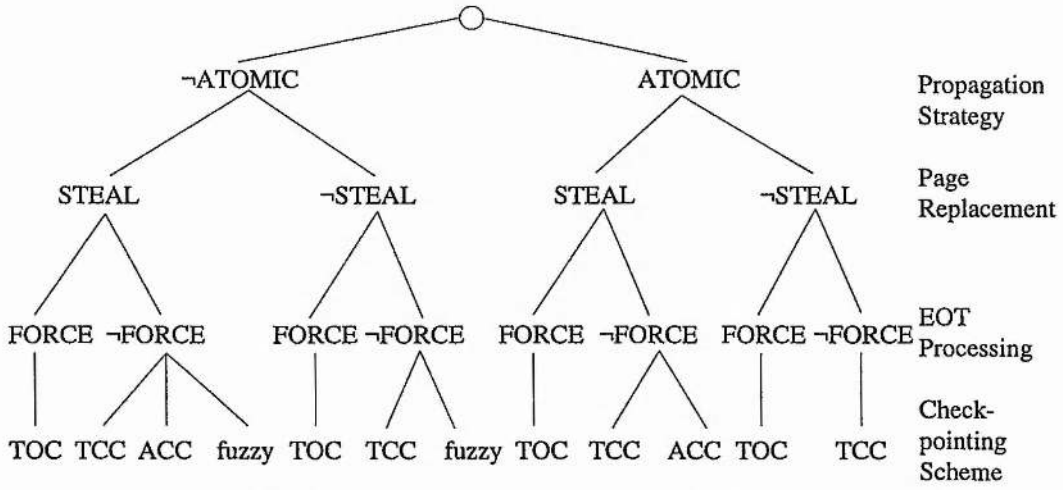


Figure 2.2: Classification Scheme for Recovery Concepts

The following sections give examples of how some of these categorisations may be realised in implementations.

2.2.3 Write-ahead Logging

Write-ahead logging mechanisms [Dav73] are the most common recovery schemes used in database systems. These mechanisms use a log file or partition to record information required to bring the database to a consistent state in the event of system failure. The term write-ahead is often used to emphasise that a record of a database update is written to the log before the update is performed.

Examples of systems that make use of logging schemes are System R [GMB+82], ARIES [MHL+92], Ingres [Sto86], Sybase, Oracle, O_2 [VDD+91], Mneme [MS88], Argus [OLS85], Eos [GAD+92], Object Design's ObjectStore [LLO+91], Exodus [FZT+92] and earlier versions of Texas [SKW92]. RVM [SMK+93] provides support for recoverable persistent virtual memory using page logging, and the Cedar file system [Hag87] makes use of logging to increase the throughput of writes and speed up recovery. There are two basic styles of logging: the write-ahead log with deferred updates and the write-ahead log with immediate updates.

2.2.3.1 Logging with Deferred Updates

In deferred update logging (Figure 2.3), a transaction's updates are written the to log and update propagation to the database is deferred until after the transaction successfully commits.

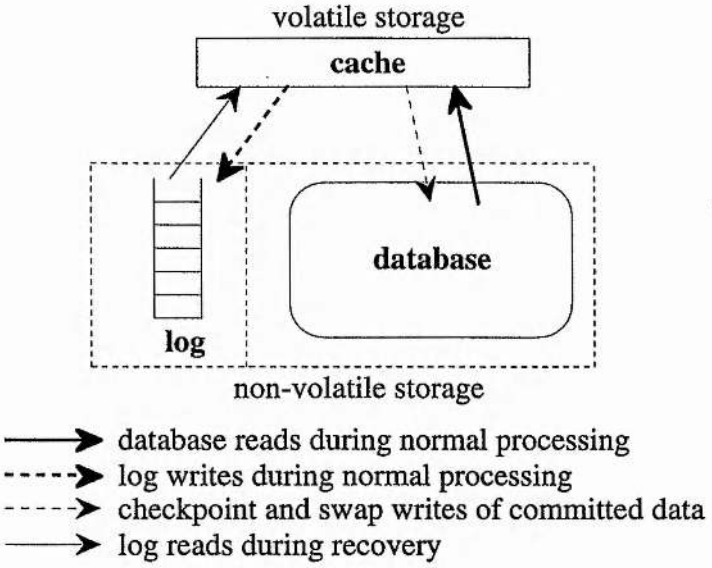


Figure 2.3: Write-ahead Logging with Deferred Updates

Each database update causes a record to be written to a log buffer. A record is composed of the updated data, the data's location in the database and the identifier of the transaction that performed the update. When a transaction commits, all update records are flushed to the log. The transaction is committed by writing a commit entry to the log. The transaction's updates are propagated to the database any time after the transaction commits.

If the system crashes after a transaction commits and before the transaction's updates are propagated to the database, the log entries are used on restart to *redo* the updates. In other words, the updates are read from the log and written to the materialised database. The recovery process is idempotent since database updates from the log may be performed a number of times with the same result as if the updates are performed once. If a transaction aborts or the system crashes before a transaction commits, none of the transaction's updates are reflected in the materialised database. Hence there is no requirement for *undo* operations either on restart or during rollback.

Transaction rollback involves simply discarding the transaction's updates from the cache and writing an abort record to the log. On restart this record indicates that any of the transaction's updates found in the log must be ignored.

Using Haerder and Reuter's classification logging with deferred updates is {¬*atomic*, ¬*steal*, ¬*force*, TOC/TCC/fuzzy}. Since non-committed updates are never written to the materialised database a deferred update log can be either *steal* or ¬ *steal*. Checkpointing involves updating the database with committed updates buffered in the cache and writing a checkpoint record to the log. This record indicates that committed updates held in the log are redundant. An action-consistent checkpointing scheme (ACC) cannot be used since it would involve updating the database with non-committed data that would require *undo* information in the event of a crash.

2.2.3.2 Logging with Immediate Updates

In a log with immediate updates (Figure 2.4) *before-images* are written to the log prior to writing updates (after-images) to the database. The before-images (the original values) may be required after a system crash to *undo* non-committed updates from the materialised database.

Before data is updated in the cache, the before-image of the data is written to the log buffer. The before-images must be flushed to the log before updating the database. When a transaction commits, the required before-images are flushed to the log and then the transaction's updates are written to the database. The transaction is committed by logging a commit entry to signify that the transaction's before-images should be ignored on restart.

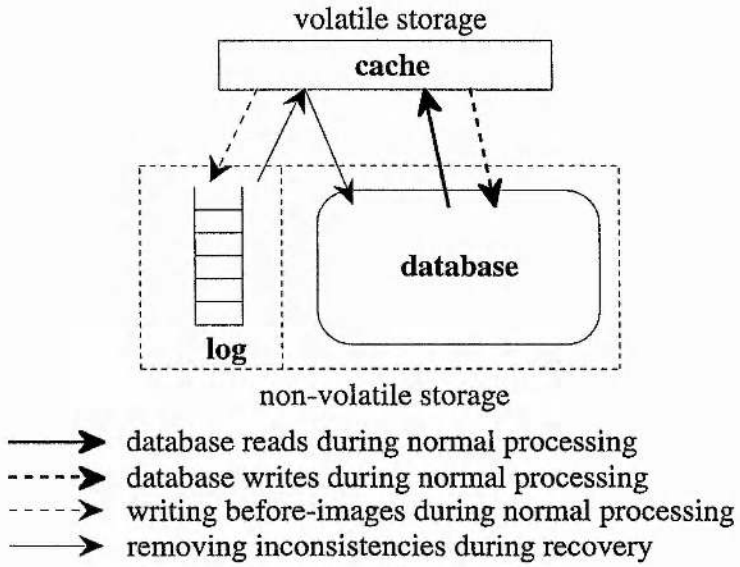


Figure 2.4: Write-ahead Logging with Immediate Updates

After a system failure the log is read backwards to find the before-images of potential inconsistencies in the materialised database, in other words those before-images that are not associated with committed transactions. The appropriate before-images are

copied to the database to remove (*undo*) potential updates made by non-committed transactions. Recovery is idempotent since *undo* operations to the database may be performed any number of times with the same result. Once all *undo* operations are complete, the log is marked as being empty to avoid performing the same operations again if another system failure occurs.

Transaction abort involves performing *undo* operations to remove any of the transaction's updates from the materialised database and writing a transaction abort record to the log. This ensures that the transaction's before-images in the log are ignored on restart. No *redo* operations are required since all committed updates are present in the materialised database. As a consequence, no checkpoints are required in immediate update logs. Using Haerder and Reuter's classification logging with immediate updates is classified as {¬*atomic*, *steal*, *force*, -}.

2.2.3.3 Undo/Redo Logging

Under some workloads, the above logging schemes may be too restrictive. For example, in immediate update logging, the overhead of writing before-images to the log during a commit, in addition to writing updates to the database, may be high. In deferred update logging, updates should ideally fit into the cache. If not, the log may be used to hold updated data swapped out of the cache, introducing the possibility of read operations on the log during normal processing to obtain the most recent version of data. A drawback is that these reads may increase the cost of seeking to the end of the log during a commit. An alternative is to use an additional area of non-volatile storage for swapping updated data, with the overhead of performing reads during a commit to copy these updates to the log.

A more flexible logging technique exists which encompasses the characteristics of both mechanisms described above. This form is known as an *undo/redo* log. In such a log, updates are written either to the log or to the database. Swap writes are normally directed to the database and before-images are written to the log for recovery. During a commit, updates are written to the log. This mechanism ensures that workloads that fill the cache with updates may be accommodated and that commit writes (to the log) are fast. If a crash occurs, inconsistencies in the materialised database are removed by overwriting them with the before-images held in the log, and committed updates held in the log are written to the database. Unlike deferred update logging *undo/redo* logs may also employ ACC checkpointing schemes since the *undo* information required for ACC strategies is available on restart. This mechanism is {¬*atomic*, *steal*, ¬*force*, any} using Haerder and Reuter's classification.

2.2.3.4 Optimising Logging

One of the distinguishing characteristics of logging mechanisms is that log writes are performed sequentially. An optimisation is to buffer log records and to perform fewer, larger writes to increase write throughput to the log. The buffer need only to be flushed to the log when the cache becomes full or if it is necessary for recovery. In deferred update logging a transaction's updates must be present in the log when the transaction commits. Hence it is possible to defer flushing the buffer to the log until the transaction commits. In immediate update logging, pinning updates in the cache enables flushing the buffer of before-images to the log to be deferred until the transaction begins to propagate its updates to the database. By writing committed updates to the database opportunistically, the cost of propagating to the database in deferred logging may be reduced.

Deferred update logs may be classified according to the type of information recorded in the log: they are either physical or logical (also called operational). The term physical logging indicates that data values are recorded in the log. The granularity of the data is normally pages or objects. Difference logging is an optimisation that may be employed in either deferred or immediate update logging. It consists of recording only the byte by byte differences between the before and after-images of data and can reduce the amount of information written to the log when compared to writing whole pages or objects. Logical logging is designed to further reduce the amount of information written to the log in deferred update logging. Instead of writing data values, high level operations performed on the database are logged. For example, inserting a tuple into a relation may cause a number of physical changes, such as updating the index and the reorganisation of data. In a physical log many records are needed to reflect these changes. In contrast, logical logging needs only record that the update takes place and to record the value of the tuple.

A further optimisation may be achieved in logging by taking advantage of the data rate mismatch between CPU and disk to perform compression on data written to the log. This may reduce the amount of log data written and hence may reduce I/O costs with the penalty of a marginal increase in CPU cost.

2.2.3.5 The Database Cache

The DB Cache [EB84] is an example of a page-based deferred update logging mechanism that aims to increase the throughput of small transactions by delaying the propagation of updated pages to the database until after commit time. During a commit, updated pages are written sequentially to a non-volatile log called the *safe*.

Figure 2.5 illustrates the layout of the DB Cache. The safe is a non-volatile storage device that permits fast sequential access and is at least as large as the cache. Pages are read from the database into the cache. Updated pages remain in the cache until the transaction commits at which time they are written sequentially to the safe. Committed pages may remain in the cache for use by other transactions, may be written opportunistically to the database, or chosen for replacement and written on demand. During the recovery process the only action required is to read the safe pages into the cache (redo). Since non-committed pages are never written to the database the mechanism is *no-undo*.

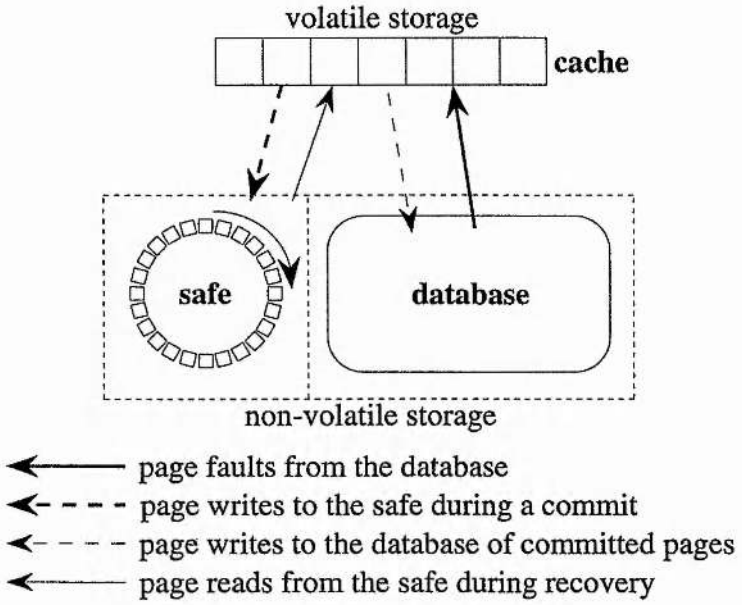


Figure 2.5: The DB Cache

Whenever the safe becomes full, pages in the safe required for recovery are flushed to the database. One of the problems therefore is determining which pages are required in the safe during normal processing and finding the pages to read into the cache during restart. During normal processing the mechanism maintains a volatile bitmap, with one bit for each page in the safe, to indicate which safe pages are required for recovery. Whenever the safe becomes full a *safe-begin-pointer* is advanced to indicate the bit corresponding to the first page required for recovery. If more free pages are required, committed pages may be flushed from the cache to the database rendering these pages in the safe *restart-free*.

During recovery, page header information is used to decide which range of safe pages hold committed pages that had potentially not been written to the database before the failure. The pages in the range are read into the cache. If two versions of the same database page are read, the older version is discarded. Once the safe has been read, normal processing resumes. It is not necessary to write the committed pages to the

materialised database, since at any point in time the database consists of the contents of the materialised database and the contents of the cache. If a system crash occurs during restart the safe is simply re-read. Hence recovery is idempotent.

The DB Cache does not write pages to the safe until transaction commit to avoid the possibility of reading pages from the safe during normal processing, and hence minimises the costs of safe write seeks during normal processing. This imposes a limitation on the number of pages that may be updated (limited to the size of the cache). Elhardt and Bayer suggest swapping updated pages to an additional area of non-volatile storage to accommodate workloads consisting of large and/or long lived transactions. The checkpointing scheme in the DB Cache is fuzzy since checkpoints are generated whenever the safe becomes full, and since only a fraction of the committed pages are propagated to the database. The DB Cache is {¬*atomic*, ¬*steal*, ¬*force*, fuzzy}.

2.2.4 Shadow Paging

Instead of using a physically separate log file, shadow paging mechanisms maintain a logical log within the database. Page replacement algorithms are used to control the movement of pages between the cache, the database and the logical log such that a consistent state is always recoverable. A page map is maintained to record the disk locations of database and log pages. The first time a database page is written to disk a shadow copy of the page is made in the log so that a before-image is always available after a system failure. A new consistent state is obtained during a transaction commit using a mechanism that atomically updates the page map so that the logical log is empty and all committed updates are visible in the materialised database. Two variations of shadow paging are discussed.

2.2.4.1 After-Image Shadow Paging

After-image shadow paging [Cha78, Lor77] ensures that updated pages never overwrite their before-images on disk - an updated page is written to a shadow copy in the logical log. A page map on disk maintains the mappings between the database pages and disk blocks. The mechanism maintains a mirrored root page from which the last consistent mappings are found. Figure 2.6 illustrates a database in which two pages are modified in the cache, one of which is shadowed in the log.

Reads operations cause pages to be faulted into the cache, where they may be updated. The first time an updated page is written to disk it is written to a free block (its shadow) in the log. Transaction commit involves flushing all updated pages to their shadow blocks. The page map on disk is then atomically updated using a

technique such as Challis' algorithm [Cha78] to reflect the new locations of updated database pages. Using Challis' algorithm the page map is described by a mirrored, timestamped root block on non-volatile storage. During a commit the older block is updated to record the new state of the page map. Using this technique the after-images of pages in the log are in effect atomically propagated to the database during a commit. The blocks holding the before-images of committed pages become redundant and may be overwritten.

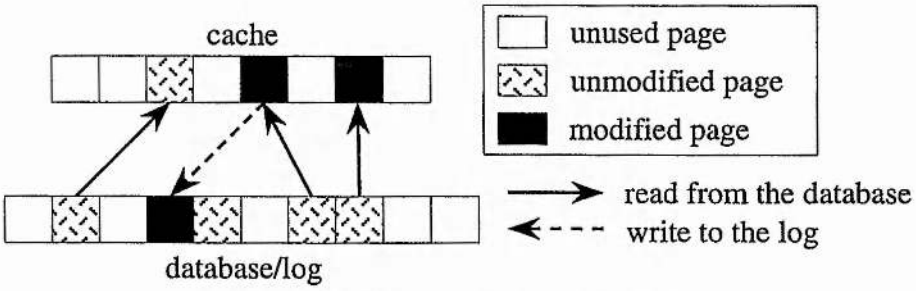


Figure 2.6: After-Image Shadow Paging

Since propagation is performed atomically, the page map on disk constitutes the state of the database at the last successful commit and so the materialised database is never in an inconsistent state after a system crash. This eliminates the need for *undo* and *redo* operations on restart, and hence by definition recovery is idempotent. Transaction abort involves discarding the appropriate updated cache pages and reverting any corresponding page mappings to their original values. Checkpointing in after-image shadow paging is transaction-oriented since propagation occurs each time a transaction commits. Using Haerder and Reuter's classification, after-image shadow paging is $\{atomic, \neg steal, force, \text{TOC}\}$.

Implementations of after-image shadow paging schemes are used in Napier88 [MBC+89, Mun93], CASPER [VKD+92, Vau94] and Gemstone [BOP+89]. Shadow paging is also used along with logging in System R [GMB+82].

2.2.4.2 Before-Image Shadow Paging

In contrast to the previous mechanism, before-image shadow paging [Bro89, BR91] always writes pages back to their original blocks. The mechanism ensures that before-images of the updated pages are available for recovery in a log appended to the end of the database. The page map on disk is used to record the locations of these shadow pages.

Before an updated page is first written to the database, the before-image of the page is written to the log. The page map on disk is updated to record the location of this shadow. Further updates to the same page need no further shadowing. Figure 2.7

illustrates a database in which two pages are modified and shadowed. One page is updated in the cache but has not yet been shadowed.

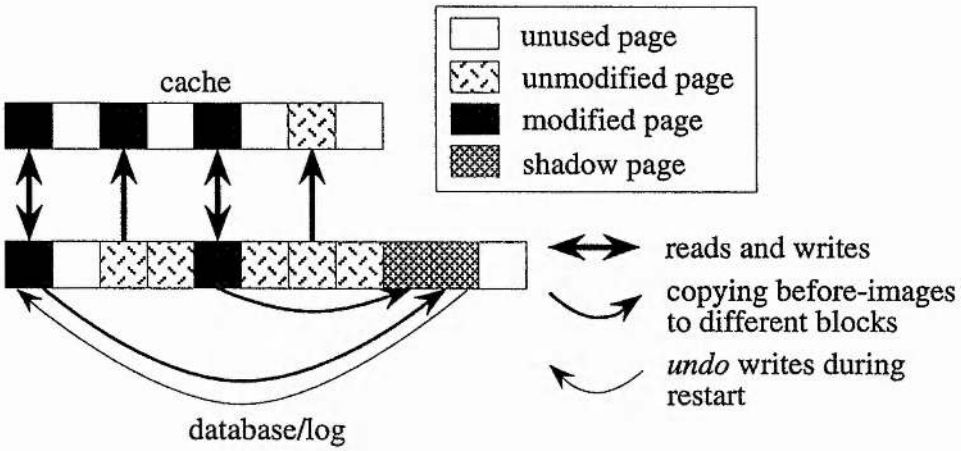


Figure 2.7: Before-Image Shadow Paging

The mechanism ensures that by the time a transaction commits, each page updated by the transaction has been shadowed and that the page map on disk has been updated to record the locations of these shadows. On commit, the updated pages are written to the database. The page map on disk is atomically updated to remove the locations of the before-images. This effectively sets the database to a new consistent state. Since updated pages are written in place, the materialised database may contain inconsistencies after a crash due to page swapping or an interrupted commit. If a crash occurs, the page map contains the locations in the log of the before-images of potentially inconsistent database pages. These pages are copied to their original locations in the database (*undo*), returning the database to the state at the last successful commit. Recovery is idempotent since before-images may be copied to the database any number of times with the same result. The mechanism is *no-redo* since all committed transactions' updates are reflected in the materialised database. Transaction abort involves discarding any cache pages updated by the transaction and overwriting any updated pages written to the database with the appropriate before-images. The page map is atomically updated to discard the transaction's before-images from the log.

Using Haerder and Reuter's classification, before-image shadow paging is {¬*atomic*, *steal*, *force*, TOC}. The mechanism is ¬*atomic* since propagation can be interrupted by system failures.

2.2.4.3 Optimising Shadow Paging

In after-image shadow paging two adjacent database pages may be located on physically distributed blocks on non-volatile storage. This may cause an increase in

seek times under workloads that access these two pages consecutively. Clustering schemes such as preallocating shadow pages in the same cylinder as the original pages [Lor77] may be employed to reduce these costs. On the other hand, since pages may be relocated on disk, dynamic reclustering of pages may be employed, thus potentially reducing read costs. For example, data may have been originally created on non-adjacent pages. If they are subsequently updated and committed together, they may be written to contiguous blocks on disk, thus potentially reducing the cost of reading these pages consecutively.

Optimisations in before-image shadow paging are similar to those which may be employed in immediate update logging. When a page is first updated a shadow copy of the page is made in the cache. This before-image need not be written to the log until the updated page is about to be written to the database. This provides opportunities for optimisations, such as flushing before-images to the log in batches or opportunistically, and reducing the frequency with which the page map is atomically updated.

2.2.5 Log-Structured Databases

Log-structured databases (LSD's) are based on the design of the Sprite Log-Structured File System [RO91]. In a LSD, the log itself acts as a repository for database pages. In other words the database is a logical collection of pages within the log. The mapping of the database address space onto the log is recorded by writing modified pages to the end of the log along with metadata to describe the database addresses of those pages. An atomic commit is achieved by writing a commit record to the log to specify that a new consistent state has been established. On restart, the last consistent state is found by reading all the metadata up to the last commit record in the log. These records are used to construct a transient page map in main memory to cache the locations of database pages in the log during normal processing. Log-structured databases employ either threading or compaction to manage free space on disk for new updates. Like AISP, log-structured mechanisms are {*atomic*, ¬*steal*, *force*, TOC}.

2.2.5.1 Log-Structuring Using Compaction

A compacting LSD moves live pages and metadata towards the start of the log thus reducing fragmentation and freeing up areas of the log for other updates. The log-structured persistent store proposed for Texas [SKW92] makes use of compaction. The locations of database pages in the log are held in a tree-structured page map that is itself recorded in the log. The location of the root of the tree is recorded on a known

location on disk. Atomicity of commits is attained through the atomic update of the record holding the root's location.

The locations of free blocks in the log are recorded by a bitmap generated on restart from the page map. This is searched when free blocks are required for writing updated pages. If the degree of fragmentation in the log becomes sufficiently high to degrade write performance the log may be compacted. The bitmap allows the compactor to determine which pages are live and which are free. Any one of a number of garbage collection techniques may be employed to free contiguous areas of the log.

This recovery mechanism may be viewed as a merger of logging and after-image shadow paging: updated pages are written to contiguous free blocks in the log in a similar fashion to deferred update logging and, like after-image shadow paging, metadata is used to record the locations of the latest versions of database pages.

2.2.5.2 Log-Structuring Using Threading

The potentially high cost of compacting the log may be reduced by performing incremental compaction. This may be achieved by partitioning the log into threaded fixed sized segments and performing compaction on a per segment basis. Each segment contains a number of pages and corresponding metadata to describe the database addresses of the pages. If segments become internally fragmented they may be cleaned for reuse by copying live pages into new segments.

An example of a threaded log-structuring is presented by Hulse and Dearle [HD96], and is used to provide resilient persistent processes within the Grasshopper persistent operating system [DBF+94]. The layout of a store is illustrated in Figure 2.8. The segments are threaded using *next segment* pointers and *time stamps* are used during recovery to find the last segment successfully written to the log. During normal processing, a tree-structured page map is maintained in virtual memory to record the log segment and the offset within the segment of the latest version of each database page.

When a page fault is performed during transaction processing, the page map is referenced to determine which log segment contains the required page. The whole segment is read and the page made available. When a transaction commits, the appropriate updated pages and metadata are grouped into segments buffered in the cache. When a segment becomes full it is written sequentially to a free segment in the log and is referenced by the previous segment written to the log. The commit process is completed by including a commit-complete record in the last segment written for the commit. During restart these records distinguish the pages written during

successful commits from those written by interrupted ones. During a commit the page map in main memory is updated to record the new locations of the pages in the log.

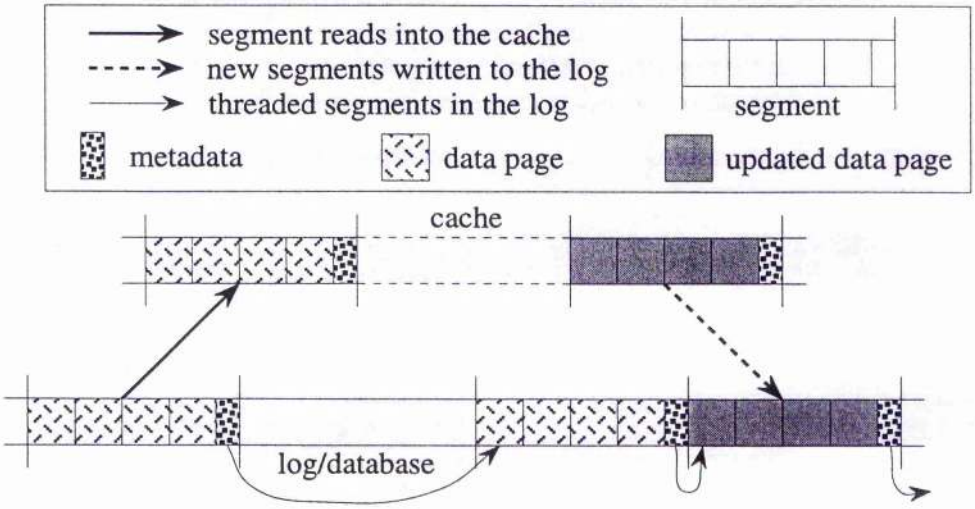


Figure 2.8: Segments in a Log-Structured Database

On restart, after a crash or after an orderly shutdown, the log is scanned forwards. The metadata in the segments are used to reconstruct the page map in main memory. Since the log may be large, this process may be optimised by occasionally writing the page map to the log during normal processing and on restart reading the latest page map from the log. Since the page map may be written lazily several commits may have occurred after the page map was last written. Therefore on restart the latest page map in the log may not constitute the latest consistent state of the database. The page map is brought up to date by reading the metadata in the segments following the last page map. The address of the latest version of the map is recorded in a known location on disk. Hulse and Dearle employ Challis' algorithm to atomically update this record.

If the log becomes full, *cleaning* is performed on internally fragmented segments in the log. This consists of reading segments into the cache and copying live pages into new segments. The mechanism discriminates between live and obsolete pages by consulting the page maps. The new segments are appended to the end of the log and the old segments are freed for reuse.

Since no operations are required to remove inconsistencies during recovery, LSD's are *no-undo*, and are *no-redo* since no operations are required to propagate committed updates to the materialised database.

2.2.6 Comments

The classification of recovery mechanisms presented by Haerder and Reuter highlights that there are numerous strategies that DBMSs can employ to provide

recovery. In order that the strategy with the lowest cost may be chosen for a particular application it is necessary to understand the trade-offs between each scheme and to be able to make accurate predictions of the costs involved within each mechanism. Such comparisons may be simplified by describing each mechanism in terms of the movement of data between a cache, a database and a logical log used to hold recovery information. For example, AISP can be described as logging in which the log is a collection of pages held on free blocks in the database. By describing each mechanism as variations of logging, the main issues concerning performance that must be addressed when comparing difference schemes for a particular application and platform are:

- How much data is read and written to the database during normal processing and checkpoints?
- How much data is transferred to and from the log during normal processing and checkpoints?
- On restart, how much data is read from the log and how much is written to the database?
- What I/O access patterns are performed?
- What are the effects on subsequent reads of the writes performed?
- Finally, what are the CPU costs incurred by the recovery manager?

This break-down of recovery costs is the basis for the new analytical cost model for recovery schemes described in Chapter 4.

2.3 Concurrency Control

Concurrent access to databases by multiple users was introduced to increase database resource utilisation. DBMSs employ concurrency control schemes to avoid inconsistencies that may result from interference among multiple users.

The most common schemes used today are implementations of the atomic transaction model [Dav73, EGL+76, Dav78]. Each read and write operation on the database is performed within a transaction. The consistency of transactions is maintained by ensuring that their interleaving is serialisable - the effects of executing transactions concurrently are equivalent to some serial execution of the transactions. An atomic transaction model is considered to be pessimistic if transactions are aborted as soon as conflicts occur, or pessimistic if transactions run to completion and are only then

aborted if conflicts have occurred. Atomic transactions are often described as adhering to the ACID properties [Gra81, HR83]: atomicity, consistency, isolation and durability.

- Atomicity, or the all-or-nothing property, refers to the organisation of the operations of a program into an atomic unit; either all the effects of the operations are visible in the materialised database or none are.

- Consistency refers to the correctness property of transactions. If a transaction is executed alone the transaction should bring the database from one consistent state to another. The system is responsible for ensuring that when correct transactions are executed concurrently, database consistency is preserved.

- Isolation refers to the fact that a transaction should perceive a consistent view of the data. For example, a transaction should not commit after having read the non-committed updates of an aborted transaction.

- Durability refers to the system's responsibility for ensuring the permanence of committed updates, in the presence of failures.

A number of attempts have been made to extend the atomic transaction model in order to increase concurrent throughput. Garcia-Molina in [Gar83], for example, proposed using semantic knowledge of operations to reduce conflicts between transactions. By studying the semantics of operations to identify which operations commute, concurrency may be improved by increasing the number of correct interleavings of transactions.

Moss [Mos81] attempts to model concurrent activities through nested transactions that structure the activities in a tree like hierarchy. A transaction hierarchy is composed of top-level transactions operating on the database, sub-transactions and atomic operations. Any transaction in the structure may call atomic operations, such as database read and write operations, and may execute sub-transactions. Leaf transactions execute only atomic operations. Transactions access copies of objects accessed by their ancestors, or copies of objects in the database in the case of top-level transactions. If no ancestor has a copy of an object, the sub-transaction obtains a copy of the globally committed version of the object from the database. Within a nested transaction, uncommitted updates may be accessed by sub-transactions. When a sub-transaction commits, its updates are inherited by its parent. When a top-level transaction commits, the updates made by the transaction and inherited from sub-transactions are committed to the database. Consistency is maintained by ensuring a serialisable schedule of reads and writes to the database by top-level transactions.

Open nested transactions proposed in [Wei86] are extensions to the nested transaction model. These models permit partial results to be viewed outside the transaction hierarchy. This is achieved by permitting a sub-transaction to commit changes to the database. If an ancestor of a committed sub-transaction subsequently aborts, a compensating transaction associated with the sub-transaction is executed to reverse its effects in the database.

For some database applications the ACID properties can be too restrictive. In modern CAD/CAM applications, for example, users may wish to co-operate to update a shared design before coming to a mutual agreement to commit the changes. This is not possible using an atomic transaction model due to the isolation property. Another drawback of atomic transactions is that they may restrict potential concurrency in systems executing long-lived transactions. These transactions involve either access to large amounts of data or involve long delays in their execution. The serialisability constraint of atomic transactions may cause long delays for other transactions wishing to access the same data. Furthermore the isolation property may lead to an increase in the probability of conflicts occurring between transactions, thus increasing the frequency of transaction aborts.

The saga model [GS87] is an attempt to increase concurrency in systems that execute long-lived transactions. Transactions are broken down into a number of atomic transactions $\{T_1, T_2, \ldots, T_n\}$, the first n-1 of which are associated with compensating transactions $\{C_1, C_2, \ldots, C_{n-1}\}$. The successful completion of a saga depends on the success of the serial execution of each component transaction. The failure of a saga, caused either by system crash or by the failure of a component transaction (T_k), requires that the compensating transactions $\{C_{k-1}, C_{k-2}, \ldots, C_1\}$ are executed to reverse the globally visible effects of the committed component transactions. The model relies on the programmer being able to break the long-lived transaction down into a number of components for which compensating transactions must be constructed.

Nodine et al. [NRZ92] attempt to model co-operation by allowing sharing between co-operative activities. Activities are modelled in a nested fashion. The internal nodes are transaction groups each of which is composed of a set of members that are either other transaction groups or co-operative transactions. The members of a transaction group co-operate to achieve a single task. Consistency within a group is maintained by ensuring that all operations performed adhere to group-specific user-specified constraints. These constraints are defined using a grammar to describe the sequences of operations that must occur within a transaction group and the patterns of operations

that are forbidden. Similarly to nested transactions a member obtains copies of objects from its parent and updated objects are inherited by the transaction group.

2.4 The Flask Architecture

2.4.1 Introduction

Traditionally database systems use one model of concurrency. This may be restrictive since it does not permit concurrency to be designed to provide optimum performance for a particular application. Furthermore such a system cannot accommodate applications that require different models of concurrency. The Flask architecture [MCM+94] uses a more flexible approach to provide the appropriate model of concurrency for the application. This is achieved by separating out the issues of concurrency from other DBMS components thus allowing a number of models to be implemented.

2.4.2 The Flask Framework

The framework of the Flask architecture is shown in Figure 2.9 as a "V-shaped" layered architecture to signify that minimal functionality is built-in at the lower layers. No assumptions are made by the lower layers about concurrency control and hence this leaves the implementor freedom to choose any desired concurrency scheme and implementation. For example, a particular specification may translate into an optimistic algorithm or alternatively a pessimistic one. Furthermore such an approach can accommodate different models of concurrency, such as atomic transactions or sagas.

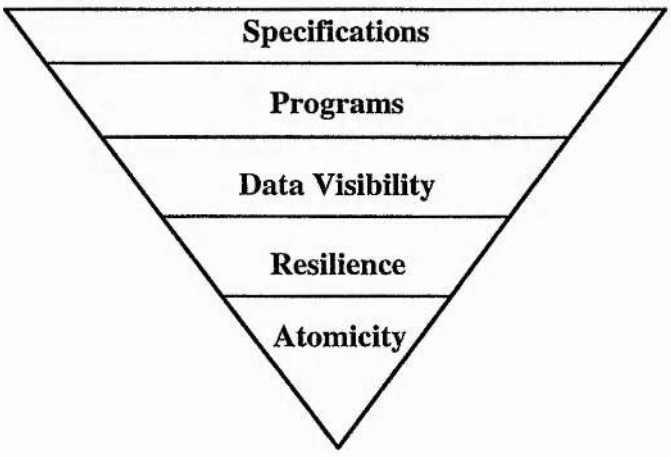


Figure 2.9: V-Shaped Layered Architecture

The architecture defines concurrency control in terms of data visibility between concurrent activities. This is reflected in the design of a conceptual layered

architecture (Figure 2.10) in which visibility is defined and controlled by the movement of data between a globally visible database and conceptual stores called access sets. Each action is associated with a local access set that isolates its view of data from all others. Actions may also use shared access sets when the concurrency model permits co-operative work between actions. Movement of data from a local access set or a shared access set to the database is through an atomic *meld* operation provided by the resilience layer of Flask. The term meld is used to describe the operation of making updates permanent on non-volatile storage and visible to other actions rather than terms like *commit* or *stabilise* since they imply specific meanings in particular models. The semantics of a meld may differ according to the concurrency model. For example, a shared concurrency model may require a number of access sets to be melded as an atomic action. Since the visibility and resilience layers make no assumptions about the higher layers the implementor is free to choose any desired scheme and implementation for the lower layers.

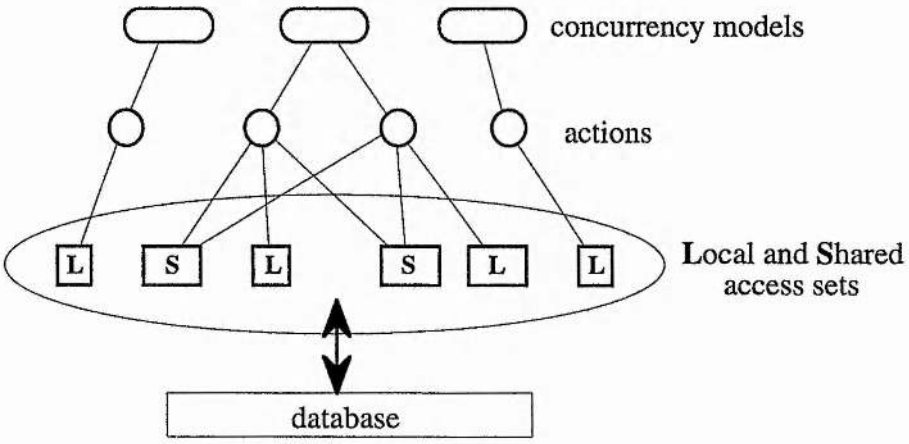


Figure 2.10: Concurrency in the Flask Architecture

The Flask architecture is designed to work with processes or actions that maintain consistency under concurrency control schemes. In general, melded changes to data do not conflict except where this happens under the control of a co-operative concurrency model. Significant events defined by a particular concurrency scheme are reported to the higher layers enabling these schemes to undertake conflict detection. This assumption frees the lower layers from the onus of interference management.

Two systems that may be used in conjunction with Flask are Stemple and Morrison's CACS system [SM92] and Krablin's CPS-algol system [Kra87]. The CACS framework provides a technique for specifying and performing concurrency control. The system does not manipulate data, but instead maintains information about its pattern of usage and indicates if operations violate the concurrency rules. CPS-algol is

an extension to the standard PS-algol system [PS87] that includes language constructs to support and manage concurrent processes. The concurrency model is essentially co-operative with procedures executing as separate threads and synchronising through conditional critical regions. Using these primitives and the higher-order functions of PS-algol, Krablin shows that a range of concurrency models can be constructed.

2.4.3 Flexible Recovery in Flask

The Flask approach to providing flexible recovery independent of concurrency control involves associating each action with a local access set that isolates its non-melded updates from other actions and from the previously melded state of the database. Actions may also use shared access sets when the concurrency model permits co-operative work between actions. Movement of data from a local access set or a shared access set to the database is through an atomic *meld* operation provided by the recovery manager. By assuming that object conflicts are detected by the concurrency control layer the recovery manager in Flask is free to provide these access sets using any suitable implementation.

In page-based recovery mechanisms these access sets may be provided by associating each action with an action page map. When an action updates a database page the action receives a copy of the page and an entry is inserted into the action's page map to record the fact. If another action updates the same database page, it receives a different copy of the most recently melded version of the page thus ensuring that the non-melded updates of the two actions are isolated. During a meld the action's page map is accessed to determine which pages must be melded. A recovery mechanism is responsible for writing these pages to non-volatile storage.

Since the meld resolution is at a page level the changes made by a melding action must be propagated to other actions' copies of the same page. Suppose that two actions A and B modify different objects on the same database page. Because of the isolation provided by per-action page copies, action A can meld without affecting B. For B to subsequently meld it must incorporate the changes made by action A. The algorithm that meld uses to propagate changes is dependent on the particular concurrency model in operation and is determined at the concurrency control layer of the Flask architecture. Under the assumption that the higher-layer performing concurrency control can detect object-level conflicts there are a number of methods of achieving this. In concurrency models that require isolation for example, in which the model requires that two concurrent actions do not modify the same object, it is possible to use logical operations for efficiency to propagate the changes.

In an atomic transaction model, changes may be propagated by performing page xor operations. Suppose two transactions A and B have changed different objects on the same page P and transaction A melds (Figure 2.11). The changes made by A to page P can be calculated by performing an xor of transaction A's version of page P onto the original version of the page, i.e. as it was at the last meld. This derives a page of changes made by A to page P. These changes are propagated onto transaction B's copy of P using a page xor operation. The meld propagation formula can be written as:

$P_B := (P_A \textbf{ xor } P_O) \textbf{ xor } P_B$

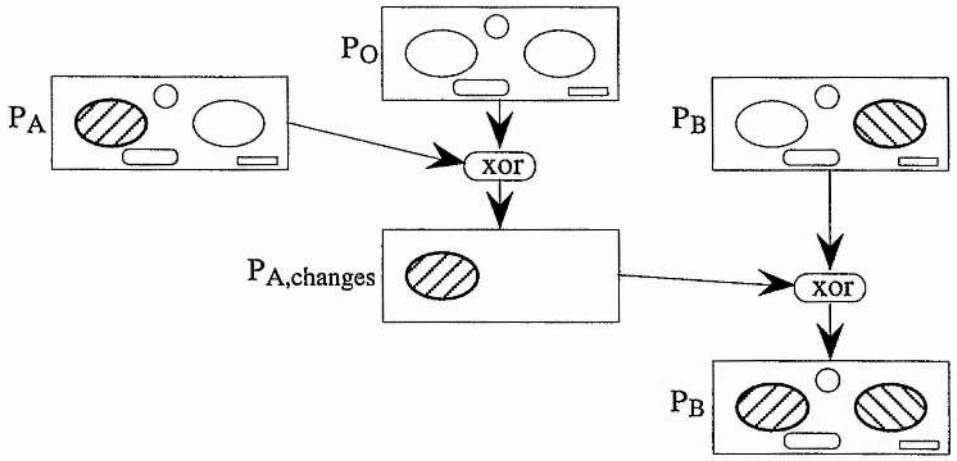


Figure 2.11: Change Propagation Using Page Xor Operations

where P_A is transaction A's copy of page P, P_O is the original version of page P and P_B is transaction B's copy of page P. Thus B's version of page P now includes the changes made by A.

Change propagation can be performed eagerly, or lazily on demand. Eager propagation is performed immediately after each action melds based on the assumption that all transactions eventually commit. Lazy propagation takes advantage of the fact that propagation is not be required until a transaction accesses the melded updates of another transaction. Lazy propagation therefore involves only performing change propagation when required. In the case above, this means that if transaction B aborts, the unnecessary propagation is avoided.

2.4.4 Concurrent After-Image Shadow Paging

The initial instantiation of Flask realises access sets through a concurrent version of after-image shadow paging [Mun93]. Figure 2.12 illustrates the layout of a concurrent after-image shadow paged (CAISP) database. A main page map on disk contains the

mappings between database pages and disk blocks, and on restart constitutes the last consistent state of the database. Each action is associated with an action page map. When an action updates a page, it receives its own copy of the page which is mapped to a free block on disk using the action's page map.

When an action melds, updated pages are written to their new blocks and the transient main page map is updated with the mappings recorded in the action's page map. The transient main page map then atomically replaces the page map on disk using Challis' algorithm [Cha78] thereby atomically propagating updates to the database. The changes made by the action must then be propagated to the pages of other actions. This is achieved using the change propagation technique described in Section 2.4.3. An action abort involves freeing all pages updated by the action and discarding the action's page map. No undo operations are required to abort an action since the original versions of the database pages are still available through the transient main page map.

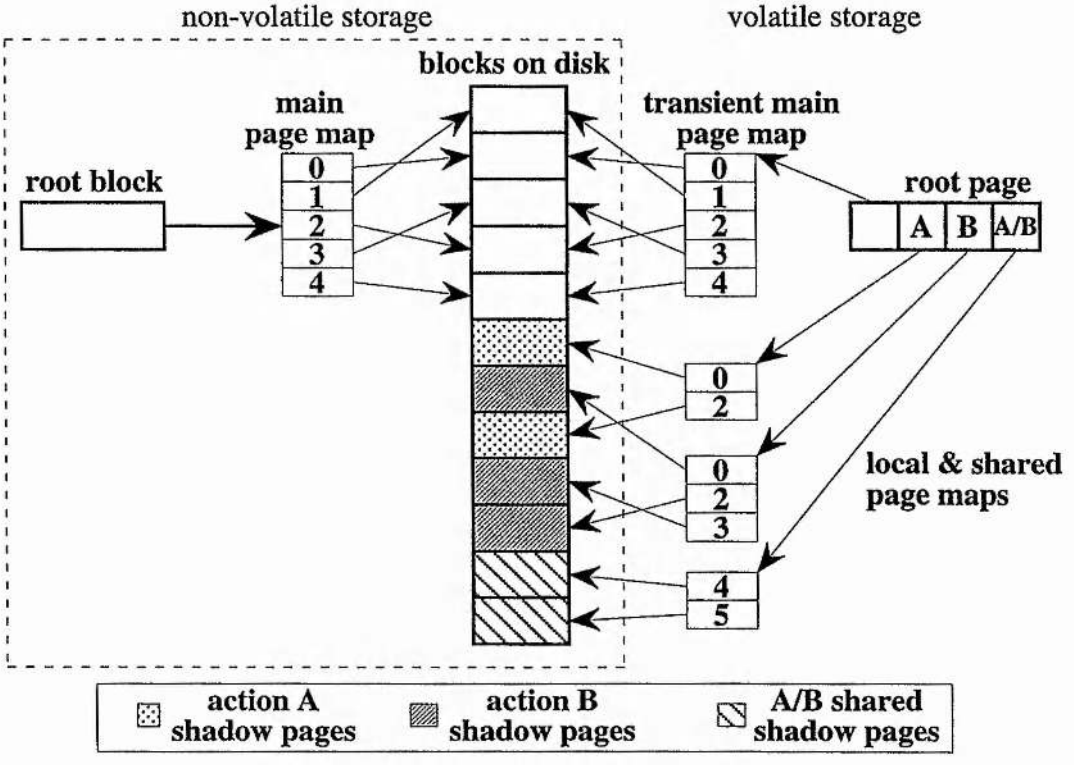


Figure 2.12: Concurrent After-image Shadow Paging

The CAISP mechanism may also be used to implement concurrency models in which non-committed updates may be shared between actions. This is illustrated in Figure 2.12 where two actions access shared copies of pages 4 and 5, although the meld actions are not defined since they are specific to the concurrency control needed.

2.4.5 Summary

Flask goes some way to providing the flexible architecture used in this thesis. The responsibility of recovery management and concurrency control are separated, thereby enabling implementations of recovery schemes to be developed and altered independently of concurrency control. The flexibility in the recovery component of Flask is achieved through an interface that places few constraints on the recovery manager and that makes no assumption about concurrency control. Hence any one of a number of recovery mechanisms may be adopted. The work presented takes advantage of this flexibility by developing two new recovery schemes to allow experimentation with different mechanisms executing the same workloads over the same data.

2.5 Analytical and Empirical Modelling

In designing and building DBMSs it is often an advantage to compare the efficiency of various designs before implementing them. One method of comparing designs is to build prototypes and to perform empirical measurement on their execution. Since this is often an unrealistic option due to the high time and labour costs required to build such prototypes, an alternative approach is to model them analytically.

2.5.1 Analytical Modelling

Analytical modelling of database systems involves developing mathematical functions to describe the behaviour of the components of DBMSs and to derive the performance of each system. The models are based on analysis of the components' designs from which a number of assumptions can be made to make the models tractable. These simplifying assumptions are required to reduce complex interactions between the many issues that must be considered when comparing DBMSs:

- the style and workload of the applications run on the DBMS;
- frequency of system failure and transaction abort;
- the platform configuration;
- interactions among other DBMS components.

An analytical model for comparing recovery mechanisms is presented in [Reu84]. The model calculates the transaction throughput of each mechanism under a particular workload based on the potential number of I/O block transfers (*availability interval*)

that may be performed in the mean time between failures. The model takes into account various aspects of the workload, recovery mechanism and platform:

Workload:

- number of I/O operations performed to process each transaction;
- ratio of update transactions to read-only transactions;
- inter-transaction temporal locality, i.e. probability that an accessed page is still in the cache after being accessed by a recent transaction;
- probability of transaction abort;

Recovery Mechanism:

- frequency of checkpoints required by mechanisms;
- overheads of transaction rollback and recovery;
- overheads of maintaining page tables;

Platform:

- size of the cache.

For each mechanism, mathematical models are developed to calculate the average number of I/O operations required to process a transaction. Models are also produced for each recovery mechanism to calculate the proportion of the *availability interval* required for transaction rollback, checkpointing and recovery. The remaining I/O operations in the *availability interval* are divided by the average I/O operations required for a transaction. This results in the average transaction throughput between failures of each mechanism. Altogether ten recovery mechanisms are analysed and compared. The mechanisms are split into three groups with the following properties:

page-level logging

¬atomic	steal	¬force	TCC (only at system shutdown)
¬atomic	steal	¬force	ACC (at regular intervals)
¬atomic	steal	force	TOC

object-level logging

¬atomic	steal	¬force	TCC (only at system shutdown)
¬atomic	steal	¬force	ACC (at regular intervals)
¬atomic	steal	force	TOC

miscellaneous

¬atomic	steal	¬force	fuzzy
atomic	steal	¬force	ACC
atomic	steal	force	TOC
¬atomic	¬steal	¬force	fuzzy

From evaluations of the cost models using different transaction workloads, Reuter concludes that page-logging is generally more costly than object-level logging, that an increase in shared pages makes all *force* algorithms drastically worse than others and that schemes that use indirect mapping, such as after-image shadow paging, impose extra overheads unless the page-table costs can be reduced.

Agrawal and DeWitt [AD85] introduce an analytical model used to investigate the relative costs of object logging, shadow paging and differential files, and their interactions with locking, time-stamp ordering and optimistic concurrency control schemes. Rather than produce costs based on transaction throughput their model uses a performance metric that describes the burden imposed on a transaction by a recovery mechanism and a concurrency control scheme. The model incorporates CPU costs and the impact that the concurrency control schemes may have on the probability that a transaction will run to completion. Burden ratios for the different integrated concurrency control and recovery mechanisms are calculated and compared using sample evaluations from varying transaction workloads and database characteristics. The conclusions from these tests suggest that there is no overall best integrated mechanism but that a load that comprises of a mix of transaction sizes favours logging with a locking approach. Shadow paging performs rather poorly in their tests. However their model takes no account of synchronous costs, such as checkpointing in logging.

A number of assumptions are made by these models, which in light of modern technology require a re-evaluation of analytical modelling of recovery mechanisms. For example, Agrawal and DeWitt assume that shadow page table reads are read from disk, whereas with modern memory sizes the entire shadow page table may reasonably be assumed to reside in main memory. Furthermore, both [AD85] and [Reu84] assume uniform disk I/O costs, making no allowance for the different costs

of sequential, asynchronous or synchronous unclustered I/O [OS94]. Modern recovery mechanisms are specifically designed to take advantage of the differences between these costs and therefore these variations should be taken into account when modelling the costs of mechanisms.

2.5.2 Empirical Analysis

In contrast to the analytical models described above, the Predator project [KGC85] takes an empirical approach to comparing recovery methods. Prototype databases supporting different recovery mechanisms are constructed on stock hardware together with a database transaction simulator used to produce experimental workloads. A suite of transaction experiments that vary locality of update, abort frequency and I/O access methods is carried out over databases supporting concurrent shadow paging and page-based logging. The performance metrics are based on transaction throughput and mean response time. The experiments are constructed from short transactions on a small system and conclude that shadow paging works best when there is locality of reference and where the page table cache is large, otherwise logging is the better mechanism. An interesting observation made is that the transaction abort rate has a greater effect on the performance of logging recovery schemes than on shadow paging.

2.5.3 Benchmarking

Objective empirical comparisons of DBMSs may only be performed if the same application workload can be executed on all of the systems. This is not always possible since real database applications can be large and complex and hence difficult to transfer from one system to another. A solution is to develop benchmarks that are sufficiently simple to implement on a range of DBMSs and allow various aspects of the systems to be measured. Benchmarks take the form of a database and a suite of queries designed to produce workloads typical of database applications. The results measured while running the queries allow the performance of components of different DBMSs to be compared. Two commonly used benchmarks, OO1 and OO7, are used to provide workloads in Chapters 4 and 5.

2.5.3.1 OO1

The OO1 benchmark [CS92] attempts to measure the operations expected in engineering applications such as CAD/CAM. The benchmark executes on three sizes of database consisting of small parts and connections between them. Each part has eight fields: a part id, a type, an (x,y) integer pair, a build date and three out-going connections to other parts. Each connection has a type and a length. To provide some

notion of locality the connections to other parts are chosen so that each connection has a 90% chance of referencing a nearby part. The benchmark consists of three queries:

`lookup`: A set of random part identifiers is generated. The parts are fetched from the database. For each part, a null procedure is called.

`traverse`: The parts connected to a randomly selected part are recursively traversed to a specified depth. A null procedure is called for each part traversed.

`insert`: A transaction inserts a number of new parts into the database, connects each new part to three other (randomly selected) parts and commits.

The operations are executed over the database a number of times to measure response time and caching effects.

2.5.3.2 OO7

The OO7 benchmark [CDN93] is designed to provide performance metrics for comparing various components of OODBMSs, in contrast to OO1 which compares the performances of entire systems. The OO7 database consists of five types of interconnected objects, ranging in size from small *atomic parts* (similar to the parts used in OO1) to large *manuals*. The database characteristics are parameterised to allow databases of various sizes to be generated.

The benchmark is composed of queries aimed to test a number of performance characteristics including pointer traversal speed, update efficiency and the performance of the query processor (in systems where this is applicable). The queries come in three categories:

- traversals of the object graph: the traversals vary in the number and locality of the objects traversed, and whether or not updates are performed.
- queries: these are read-only database queries.
- structural modifications: one program inserts a number of new parts into the database and another deletes the parts.

Results are taken from running each query on a 'cold and 'hot' system. A cold system is one in which no data is cached, resulting in a high number of data faults. A system is said to be hot if data is cached, and results in fewer faults.

2.6 Conclusions

The cost of recovery in DBMSs not only involves the cost of bringing the database to a consistent state after failure, but also the overhead incurred in recording sufficient information in the log during normal processing to ensure that data can be recovered to some consistent state. This chapter has given background and optimisations of various recovery mechanisms used in database and persistent systems. Traditionally, DBMSs have a fixed notion of recovery and concurrency control, and have these components embedded into the system thus providing few opportunities to configure the components to a particular application. An outline of the Flask architecture was discussed to give an insight into how recovery and concurrency may be separated in a DBMS to provide the flexibility to configure each component individually.

This chapter also includes summaries of early analytical and empirical studies of DBMS components that could have been used to guide in the configuration of Flask. A consistent conclusion made from these studies is that there are significant variations in the costs of recovery mechanisms and that no one mechanism provides the best performance for all applications. Current trends in application styles, hardware configurations and operating systems weaken many of the assumptions made by these studies, and as a result the validity of past analysis may be questioned. The Chapter 4 introduces a new analytical model that takes into account modern platform characteristics and application styles in costing recovery mechanisms. A strength of the model is that it is validated by analysis of benchmarks executing over the mechanisms modelled. This is achieved by using the flexible recovery manager of Flask to allow the same application workloads to be executed over different recovery mechanisms. The following chapter develops two new mechanisms used in the Flask architecture.

3 Flexible Recovery

3.1 Introduction

In the work presented, a new cost model for recovery mechanisms, called MaStA is developed. The model is designed to predict the mechanism with the lowest cost for a given application and platform, within a flexible database system such as Flask [MCM+94]. To illustrate the use of the model in Flask and to verify the accuracy of cost comparisons of recovery mechanisms, the same workloads must be executed using a number of configurations of the architecture. At present, Flask incorporates only one recovery scheme, namely concurrency after-image shadow paging.

This chapter extends Flask with a flexible recovery manager. The manager is composed of a number of components, each of which is responsible for an aspect of recovery such as restart or page replacement. Three recovery schemes are then developed using the flexible recovery manager:

- an after-image shadow paging mechanism (AISP);
- a log-structure mechanism (LSD);
- and a log-based mechanism called DataSafe.

A detailed description of DataSafe is provided, followed by summaries of the implementations of AISP and the LSD. These particular mechanisms are chosen to emphasise that MaStA can be used to predict the relative costs of mechanisms that perform similarly (AISP and the LSD), and to provide a mechanism (DataSafe) that has significantly different I/O characteristics to those of the other schemes. For example, DataSafe employs a fixed placement policy whereas AISP and the LSD perform dynamic reclustering. The variations in the characteristics of these mechanisms are highlighted when they are compared using Haerder and Reuter's classification [HR83]: DataSafe is $\{\neg atomic, \neg steal, \neg force$, fuzzy$\}$ whereas AISP and the LSD are $\{atomic, \neg steal, force$, TOC$\}$.

The provision of these mechanisms within Flask provides an experimental base in later chapters on which the MaStA model may be validated, and also provides an opportunity to illustrate the effectiveness of the model in selecting between mechanisms for given applications.

3.2 The Flexible Recovery Manager

The flexible recovery manager is configured with different placement and replacement algorithms to provide various recovery mechanisms as illustrated in Figure 3.1.

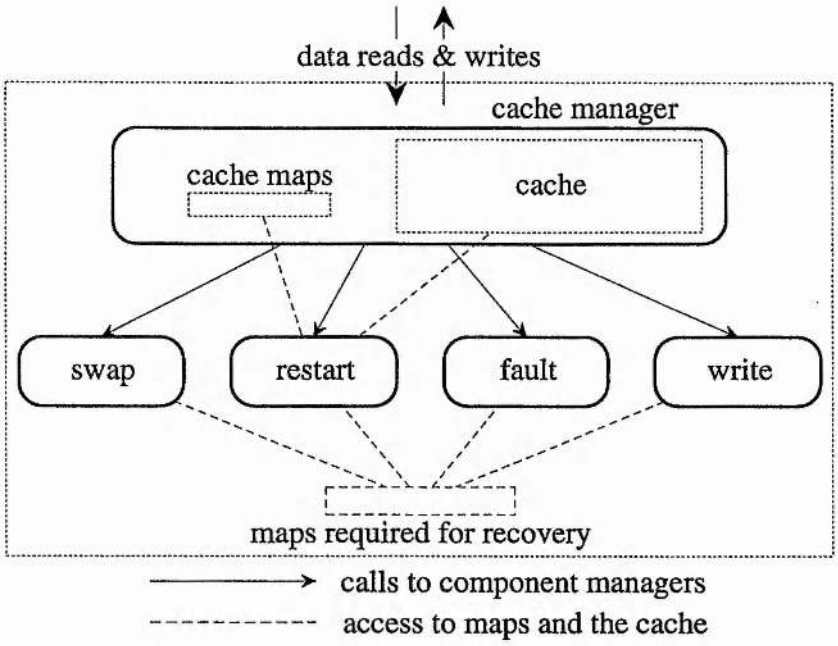


Figure 3.1: The Flexible Recovery Manager

The cache manager has a fixed interface to which all read and write operations performed on the database are directed. It provides a database cache and any maps required to translate database addresses into cache addresses. Four configurable component managers are called by the cache manager:

- The fault manager is called when data not already in the cache is accessed, and is responsible for locating the data on disk. For example, in after-image shadow paging this requires a page map to be indexed to obtain the disk locations of database pages.

- The write manager takes a cache location, a database address and the length of the data, and writes the data to disk. The disk location written to is chosen according to the algorithm defined in the design of the recovery mechanism. For example, in a deferred object logging mechanism updated objects are written to the end of the log. The write manager is also responsible for atomically updating the state of the database. In AISP, for example, this involves atomically updating the page map on non-volatile storage.

- The swap manager is called whenever data in the cache is read or updated. Calls to the swap manager enable it to collect information about the usage of cached data so that any cache replacement algorithm may be implemented. When the cache becomes full the swap manager is called to select data for replacement. The swap manager is responsible for swapping updated data to disk if required.

- The restart manager is responsible for initialising any maps required by the fault, write and swap managers. This manager is also responsible for ensuring that the materialised database is brought to a consistent state after system failures. To achieve this the restart manager is given access to the cache and the cache maps. For example, in DataSafe the cache is reconstructed on restart.

A strength of the flexible recovery manager is that it provides opportunities to modify existing mechanisms implemented in the manager, and to develop new ones, simply by replacing component managers instead of implementing entire new mechanisms. This is illustrated in Section 3.5 where the LSD mechanism is developed from the implementation of AISP by simply replacing the write manager.

Three page-based recovery mechanisms are implemented to perform the validation procedures: AISP, LSD and a new mechanism called DataSafe. The recovery mechanisms are developed by instantiating the flexible recovery manager with the same cache manager but with different swap, write, restart and fault managers. The cache manager maintains a database cache held in main memory. The same-sized database cache is used for each recovery mechanism in the framework to simplify the analysis of the I/O behaviours of the workloads. A summary of the configurations of the recovery mechanisms used is included in Appendix A.1.

3.3 The DataSafe Recovery Mechanism

3.3.1 Introduction

The DataSafe recovery mechanism [SCM+96] is based on the DB Cache [EB84]. The DB Cache is chosen as a basis for an alternative mechanism since its characteristics vary widely with those of AISP. For example, the DB Cache uses a contiguous circular log while AISP intersperses log and database pages. Furthermore AISP imposes a reclustering policy on database pages whereas the DB Cache does not.

The DataSafe recovery mechanism, in contrast to the DB Cache, is designed to adhere to the interface of the Flask recovery manager so that the independence between

concurrency and recovery is maintained. This is achieved in a similar manner to the CAISP mechanism [Mun93], through the provision of access sets.

DataSafe ensures the recoverability of a database by controlling the movement of pages of data among three areas of storage: the database, a safe and a cache. The layout of the mechanism is illustrated in Figure 3.2.

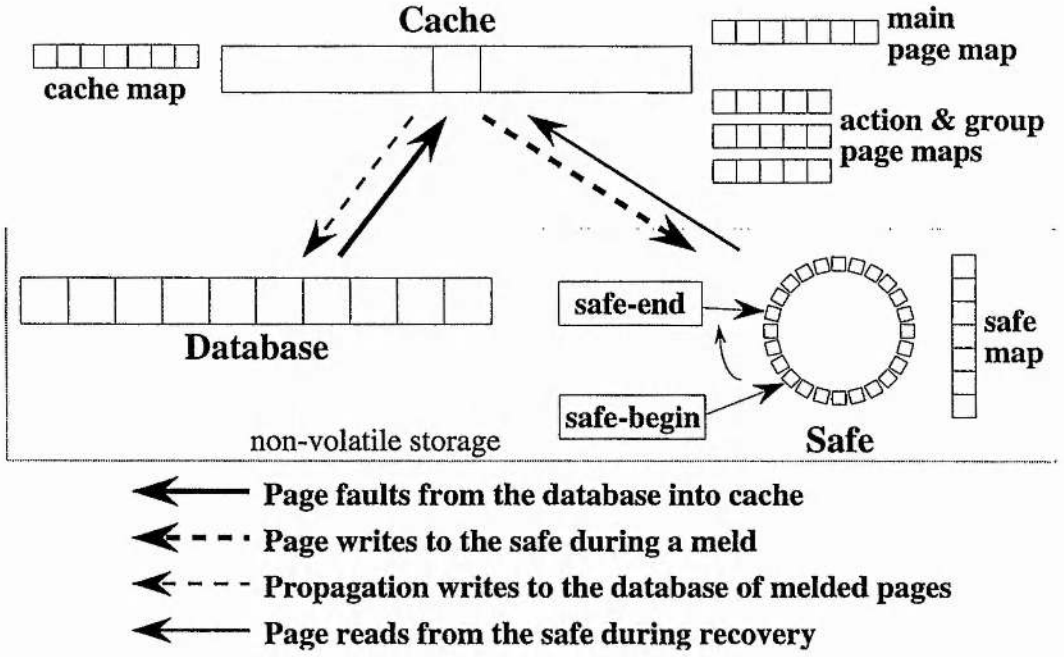


Figure 3.2: Layout of the DataSafe

Reads and writes operate on data in the cache, faulting database pages into free cache pages as required. The pages updated by an action remain in the cache at least until the action melds or aborts, under the assumption that the cache is sufficiently large to hold all updated pages between melds. Updated pages are not swapped to the database to ensure that no non-melded updates are present in the materialised database after system failures.

A meld operation involves writing the cache pages updated by the melding action to contiguous pages in the safe. This ensures that in the event of a system failure melded cache pages that have not yet been written to the database are recoverable. If insufficient free pages are available in the safe to complete a meld, safe pages that are required for recovery are written from the cache to the database. This means that they are no longer required for recovery in the safe and as such may be overwritten during the meld. After a successful meld the melded pages either remain in the cache to be reused or are propagated to the database opportunistically.

If there are no free cache pages available to fault a database page a cache page that has been melded or is unchanged is selected for replacement. A selected cache page may have been melded to the safe but not yet propagated to the database in which case it is written to the database before being replaced. This ensures that page faults operate on the database rather than on the safe.

During recovery the safe pages required for recovery are read from the safe into the cache after which normal processing resumes. This strategy ensures that no read faults operate on the safe and hence all writes to the safe may incur low sequential seek costs.

A number of maps are maintained to record information about database, safe and cache pages:

- A cache map in volatile storage records the state information of cache pages (free, original, melded or updated).
- The main page map records the cache locations of faulted database pages that have not been updated in the cache, i.e. cache pages that are duplicates of pages in the database.
- A safe map on disk records the state information of the pages in the safe.

During normal processing only a fraction of the safe contains pages required for recovery. The location of this area is recorded by a safe-begin-pointer and a safe-end-pointer held on disk.

3.3.2 The Safe

The safe is designed as a circular buffer to enable writes to the safe to be performed sequentially. The safe must be at least as large as the cache to ensure that all pages updated in the cache may be written to the safe. Since the same page may be updated and melded to the safe many times the safe may contain more than one version of a database page. Only the latest version of a page in the safe is required for recovery and then only if the corresponding cache page has not yet been propagated to the database. Thus a safe page is free if the corresponding cache page has been written to the database or if a more up-to-date version of the page is in the safe. The database locations of pages in the safe are recorded in the safe map which is written atomically to disk during each meld (see Section 3.3.4).

3.3.3 The Cache

DataSafe's swap manager holds an action's updated pages in the cache at least until the action melds or aborts. This avoids the need to maintain *undo* information since non-melded updates are never swapped to the database. The cache is designed to fit into main memory to avoid operating system page swapping. It is composed of a number of page sized frames that are empty or contain pages of data. Cache pages are tagged using the cache map as free, original, melded or updated. Figure 3.3 gives the state diagram of cache pages.

A cache page is tagged as original to signify that the page has not been updated or melded and that it may be selected for replacement if the cache becomes full. If an original page is updated by an action the update is performed on a copy of the page in the cache. Updating a copy avoids performing another fault on the database to obtain an original version of the database page should another concurrent action access the same page.

An updated cache page may have further changes made to it, become free due to an abort or be written to the safe during a meld operation. If an updated page is written to the safe the page is tagged as melded to signify that it must be written to the database before being replaced in the cache. A melded or an original page becomes free if another copy of the same database page is melded.

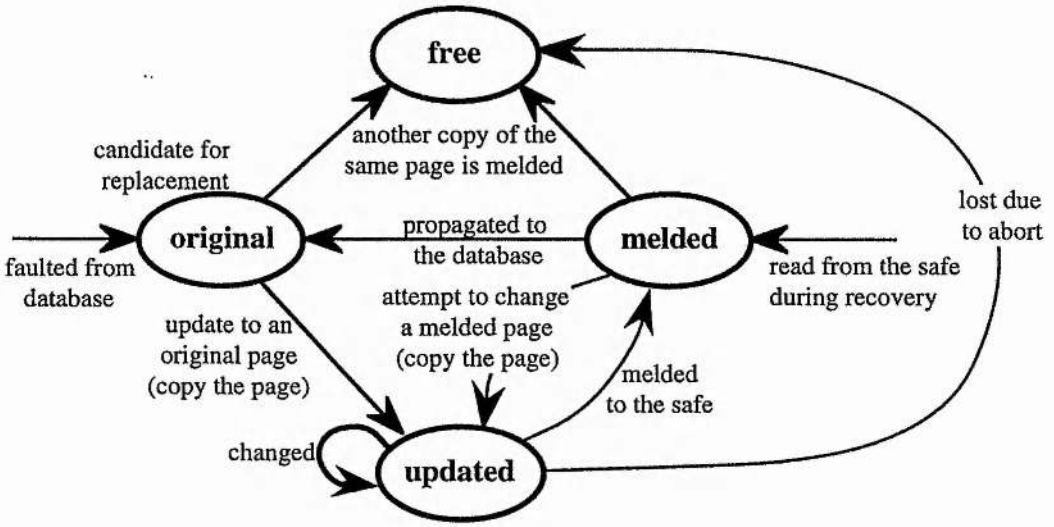


Figure 3.3: Cache Page State Diagram

If a melded page is updated by an action the update is performed on a copy of the page in the cache. This ensures that unchanged versions of melded pages are available in the cache, avoiding the need to perform propagation reads on the safe during a safe

purge should the safe become full (see Section 3.3.6 on safe purging). If a melded page is propagated to the database it is tagged as original.

Cache pages read from the safe during recovery are tagged as melded to ensure that during normal processing they are propagated to the database before being replaced in the cache.

3.3.4 Action Meld and Abort

When an action melds, the cache pages updated by the action are written by the write manager to contiguous free pages in the safe at the location given by the safe-end-pointer. The updated pages are found using the action's page map. As each page is written to the safe an in-memory copy of the safe-end-pointer is advanced and an in-memory copy of the safe map is updated to record the database location of the safe page. Any other melded or original version of the database page present in the cache becomes obsolete and is designated free in the cache map. The main page map is then updated to record the cache location of the newly melded version of the database page.

Once all the required pages have been written to the safe, the safe map and the safe-end-pointer are written atomically to disk. The safe map is updated using Challis' algorithm [Cha78]. The root page used in this algorithm records the safe-end-pointer (and the safe-begin-pointer). If a system failure occurs during a meld, all pages written to the safe by the incomplete meld are ignored on restart since the safe-end-pointer which indicates the last safe page read during restart will not yet have been updated. Atomicity of a meld is therefore attained by the atomic update of the safe map and the safe-end-pointer.

The safe is said to be full when there are insufficient pages between the locations given by the safe-end-pointer and the safe-begin-pointer to complete a meld. In such a case a safe purge (see Section 3.3.6) is performed to advance the safe-begin-pointer, before the meld begins, by a sufficient number of pages to allow the meld to be performed.

Once an action melds, the changes it has made must become visible to any other action that accesses the same data. DataSafe uses Flask's change propagation technique to copy the changes made by a melding action to the access sets of other actions.

An action abort involves freeing the cache pages updated by the action and discarding the action's page map. No *undo* operations are required since database updates are deferred until after a meld completes.

3.3.5 Restart

In DataSafe, updates to the database are deferred until after a meld completes. This avoids non-melded updates in the materialised database after system failure. Once a meld completes, propagation writes of melded pages to the database may be performed opportunistically. Since these writes are deferred some pages may not have been propagated to the database before a system failure. Restart involves reading into the cache the safe pages that potentially were not propagated to the database before the crash. The restart manager reads the safe-begin-pointer, the safe-end-pointer and the safe map from disk and scans the safe map to determine which safe pages to read into the cache. The database locations held in the safe map are used to reconstruct the main page map as pages are read into the cache.

The DataSafe mechanism ensures that the latest version of each page is either in the cache or in the database and thus ensures that no read faults operate on the safe during normal processing. This strategy ensures that all writes to the safe incur low seek costs.

3.3.6 Safe Purge

Safe purging is the process of propagating safe pages that are required for recovery to the database. A safe purge is performed by the write manager if there are insufficient free pages in the safe to write the pages updated by a melding action. A safe purge advances the safe-begin-pointer by a sufficient number of pages to allow the meld to complete. A sufficient number of free safe pages can always be obtained since the safe is at least as large as the cache.

Since the area of the safe containing safe pages required for recovery is bounded by the safe pointers, the safe-begin-pointer may only be advanced past safe pages no longer required for recovery. An in-memory copy of the safe-begin-pointer is advanced to the first safe page required for recovery. If there are still insufficient free safe pages between the safe pointers, the page at the safe-begin-pointer is propagated to the database and the safe-begin-pointer is advanced to the next safe page required for recovery. This process is repeated until there are sufficient free pages between the safe pointers. The safe-begin-pointer on disk is then atomically updated. This ensures that the meld does not write pages to the area of the safe indicated by the safe pointers that is read during restart should a system failure occur during the meld. The meld may then be performed.

The safe purge mechanism only propagates sufficient safe pages to the database to permit the meld to complete instead of propagating more safe pages. This strategy is

based on the assumption that during melds some pages in the safe become obsolete and therefore will not require to be propagated to the database during subsequent safe purges. If more than the required number of safe pages are propagated to the database during each safe purge unnecessary writes may be performed since some of the safe pages may have become obsolete during subsequent melds.

As mentioned previously, melded pages in the cache are not updated directly. This ensures that no propagation reads are required on the safe to propagate melded pages to the database, since the pages are still present in the cache. Therefore propagating a safe page to the database involves writing a cache page to the database.

Figure 3.4 gives an illustrated example of a safe purge and meld. The locations recorded by the safe-begin-pointer and the safe-end-pointer held on disk are shown. In this example seven updated cache pages are to be melded. Figure 3.4.a illustrates the state of the safe before the meld.

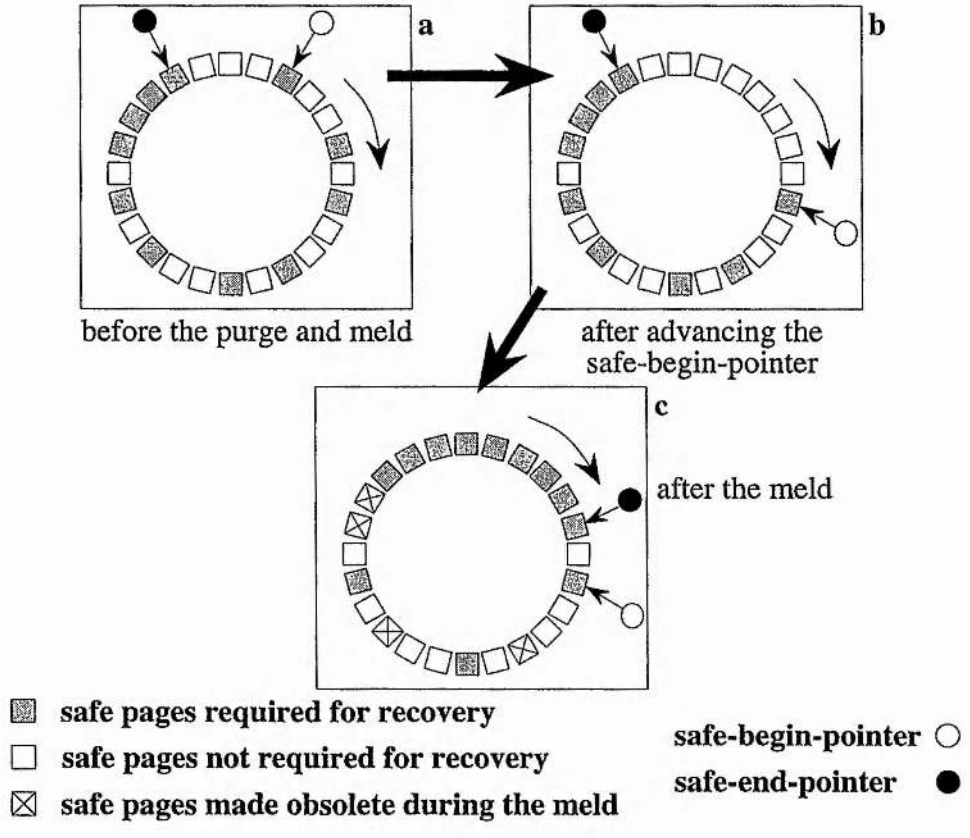


Figure 3.4: States of the Safe During a Purge and Meld

When the meld is initiated the mechanism ensures that sufficient free safe pages are available between the safe pointers to allow the meld to complete. Since in this case there are only three pages between the safe-end-pointer and the safe-begin-pointer (Figure 3.4.a) a safe purge is performed to advance the safe-begin-pointer by at least

four pages to provide at least seven free pages required for the meld. Figure 3.4.b illustrates the safe after propagating two safe pages to the database and shows the new locations recorded by the safe-begin-pointer.

The meld can now proceed. Figure 3.4.c illustrates the state of the safe after the meld completes and shows the new locations recorded by the safe pointers. The figure also illustrates that some safe pages have been made obsolete (are no longer required for recovery) due to the melding of more recent versions of those pages. This enables the next safe purge to advance the safe-begin-pointer past these pages without requiring to propagate them to the database.

3.3.7 Cache Overflow

If there are no free cache pages available to either fault a database page or to make a copy of an original or melded cache page, a cache page is selected for replacement. Only original and melded pages are replaced since updated pages must by design remain in the cache. A victim selection algorithm may give originals a higher probability of being chosen since choosing a melded page incurs the cost of propagating it to the database before replacing the page.

A potential problem of DataSafe is that the cache may become full of updated pages in which case no pages may be chosen for replacement. While this may not be a problem in some applications it is clearly a limitation for others. In such cases DataSafe may use an additional area on disk to which updated pages may be swapped.

3.3.8 Opportunistic Write Back

Since the safe ensures that melded cache pages are recoverable and they may be propagated to the database at any time. In addition to writing them to the database during a safe purge or when the cache becomes full, these writes may be performed opportunistically while no other page faults or writes are being performed. They may also be scheduled in such a way as to take advantage of the position of the disk head to reduce the seek costs incurred when writing to the database. When a melded cache page is propagated to the database the corresponding safe page becomes obsolete and so no longer required for recovery. Thus opportunistic writing of melded cache pages reduces the number of safe pages that must be written synchronously to the database by a safe purge or due to cache page replacement.

There is a trade-off between propagating melded cache pages to the database opportunistically and writing the pages synchronously during page replacement or safe purges. An opportunistic propagation policy may be adopted under the

assumption that melded pages are eventually propagated to the database through page replacement or safe purging, and by performing these writes asynchronously the overall cost of writing to the database is reduced. On the other hand by adopting a pessimistic propagation policy in which melded pages are only written to the database when required, some melded cache pages may become obsolete thus avoiding some propagation writes that would have been performed in an opportunistic policy. The decision as to which strategy to adopt is a matter policy and may be based on the characteristics of the workload executed on the mechanism.

3.4 After-Image Shadow Paging

The AISP mechanism is implemented by making use of the cache manager used in DataSafe and providing a new swap, fault, write and a restart manager to make the recovery manager behave in the manner described in the AISP design described in Section 2.2.4.1.

The restart manager reads from disk the main page map used by the fault, write and swap managers to locate database pages on disk. The restart manager initialises a disk block bitmap used by the write manager to locate free blocks. Before a cache page is written to disk the write manager accesses the main page map to determine if the database page has already been shadowed. If not the page is mapped to a free disk block (shadowed). The page is then written to its shadow block. Free blocks required for shadowing are allocated from within the database before new blocks are allocated at the end of the database. During a meld the page map is written atomically to disk using Challis' algorithm.

The swap manager is similar in design to the DataSafe swap manager with modifications to the page replacement strategy to allow non-melded updates to be selected for replacement. If an updated page is selected for replacement the page is shadowed if required and written to its shadow block.

3.5 Log-Structured Database

The design of the LSD mechanism is similar to the AISP mechanism described in Sections 2.2.4.1, with two modifications. The first is that free blocks are allocated contiguously, at the end of the log. Once the log fills, the search for free blocks starts at the beginning of the log. This design means that more pages are written sequentially to the log in the LSD than in AISP. The second modification is that updated page map pages are also written to the end of the log instead of writing them to preallocated shadow blocks as in AISP, thus potentially reducing the cost of updating the page map.

Due to the similarities of the designs of AISP and the LSD mechanisms, the LSD is implemented by simply reusing the fault, swap and restart managers used in AISP and developing a new write manager to allocate shadow pages contiguously at the end of the log.

3.6 Conclusions

The Flask architecture provides the opportunity to independently configure concurrency and recovery to suit the application. By assuming that a higher layer of Flask is responsible for concurrency control the recovery manager has the flexibility to select the mechanism that provides optimum performance for a given application without the need to perform interference management.

The first instantiation of the Flask architecture made use of a concurrent version of after-image shadow paging. The flexibility of the architecture is highlighted by developing two alternative mechanisms, DataSafe and the LSD, either of which may be interchanged with CAISP at link time. DataSafe is based on the design of the DB Cache with alterations that ensure that the mechanism adheres to the Flask recovery manager interface. In place of page header information, the DataSafe makes use of a safe map to record the state of safe pages, and in accordance with the Flask recovery interface, avoids page locking through the provision of access sets and the use of the meld propagation scheme employed in CAISP. The LSD mechanism is effectively CAISP with alterations to the shadow page allocation strategy to make log writes behave similarly to those of a log-structured mechanism.

The next chapter introduces a new analytical model for recovery mechanisms, designed to be used to select the mechanism that incurs the lowest cost for a particular application and platform. The Flask architecture, together with the three recovery mechanisms developed, provides an experimental basis in the following chapters on which the model is validated. The validation strategy makes use of the flexible recovery manager to execute the same workloads on different mechanisms.

4 An Analytical Model for Recovery Mechanisms

4.1 Introduction

This chapter presents a new analytical cost model for recovery mechanisms called MaStA [SCM+95a, MCM+95]. The model focuses on estimating the I/O overheads of recovery, taking into account the cost variations between different I/O access patterns. An analytical technique is chosen since this form of modelling is believed to be less expensive to develop than simulations or empirical measurement. The model is designed to provide a framework for comparing the costs of recovery mechanisms under a variety of different workloads and configurations, and may be used to guide the choice of mechanism for a particular application in a flexible architecture such as Flask.

The design of MaStA is based on the observation that all mechanisms may be viewed as variants of logging differing in the patterns and the number of I/O operations required to read and write data and recovery information. This design simplifies the modelling and comparing of recovery mechanisms by abstracting over the details of each mechanism and calculating their costs according to the movement of data between a database, a cache and a log during normal processing and checkpointing.

MaStA focuses only on the I/O costs of recovery mechanisms - the CPU costs are omitted. This omission is based on the assumption that I/O costs are the significant factor in the difference in performance of any two recovery mechanisms. Furthermore trends in hardware performance suggest that CPU speeds are increasing more rapidly than disk speeds, which will further reduce the significance of CPU costs when comparing mechanisms. An outline of the MaStA I/O cost model is provided, followed by a detailed discussion of how models of recovery mechanisms are constructed, and how MaStA may be applied to compare the costs of mechanisms.

4.2 Overview of the MaStA Model

MaStA categorises I/O operations performed by recovery mechanisms by the manner in which they operate. For example, a mechanism may perform data reads on the database and data writes to the log, both of which are categorised for that recovery mechanism. For the purpose of analytical modelling, these categories are termed I/O cost categories and the overall cost of a mechanism is the sum of the costs of its constituent I/O cost categories (Figure 4.1).

$$\text{Total Cost} = \sum_{i} \text{CatCost}(i),\ (i \in \text{Categories})$$

Each category is assigned one or more I/O access patterns according to the properties of the I/O operations performed by the mechanism within the category. For example, log writes may be assigned sequential write costs in a log-based mechanism. The number of accesses incurred in a category of a particular access pattern is derived from a workload function composed of workload variables such as the number of reads and writes performed by the application and locality. The cost of an I/O cost category is a product of the number of accesses of a given pattern and the cost of the pattern, or the sum of a number of such products.

$$\text{CatCost}(i) = \sum_{j,k} n_{i,j} \times A_k, \; (j \in \text{Occurrences}, k \in \text{Access Patterns}, i \in \text{Categories})$$

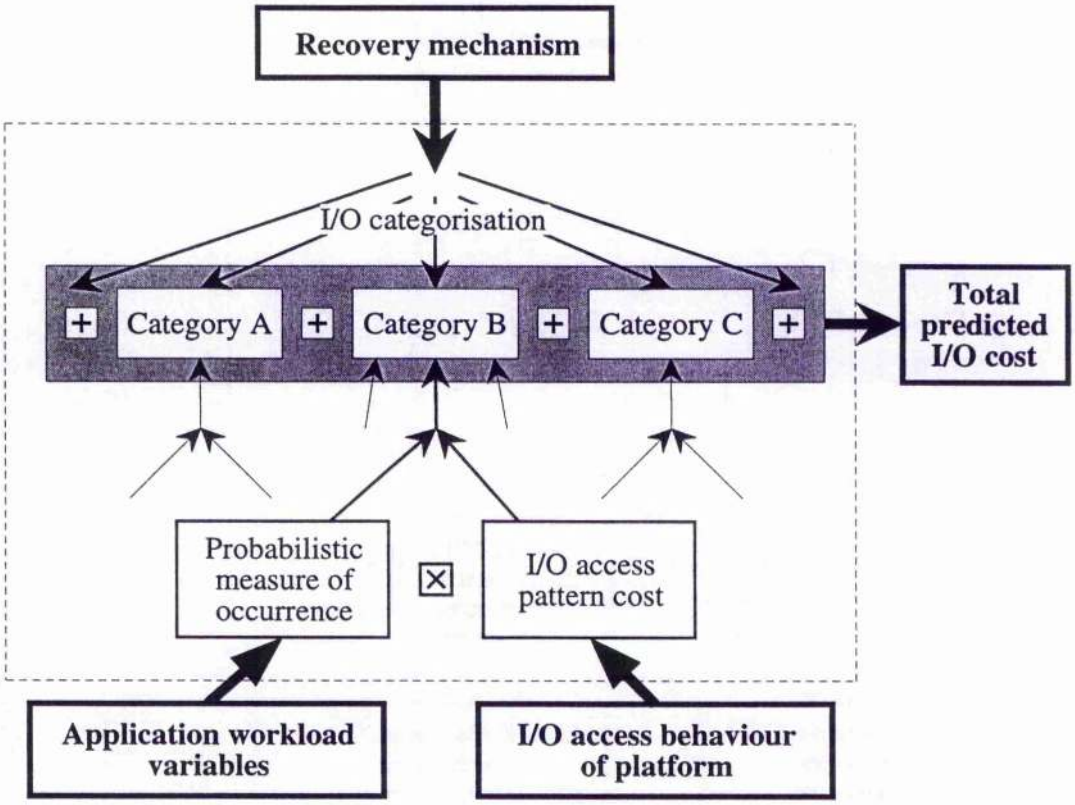


Figure 4.1: An Overview of MaStA

Remembering that this is an analytical model, the derivation of a cost estimate for a particular combination of mechanism, configuration and workload is derived by analysing:

- The workload: measuring and choosing values to predict the workload.
- The mechanism: attributing costs to each cost category by calculating the number of accesses from the workload abstraction, and assigning access patterns.

- The configuration: determining the cost of each access pattern for each platform experimentally or analytically.

4.3 Developing the MaStA Cost Model

4.3.1 Recovery Mechanisms

To illustrate the MaStA model, four page-based recovery mechanisms are examined: DataSafe; after-image shadow paging (AISP); before-image shadow paging (BISP); and a log-structured database (LSD). Summaries of the mechanisms are provided here with more detailed descriptions given in Sections 2.2 and 3.3.

DataSafe records changes in a log called the safe and updates to the database are deferred until after commit. Database updates do not move database pages so the original clustering is maintained. Updates are eventually propagated to the database opportunistically or during normal shutdown. Propagating a committed page requires a propagation write to update the database, though multiple changes to the same page by a number of transactions may result in only a single write.

In AISP a page replacement algorithm controls the movement of pages between cache, the database and the log such that recovery will always produce a consistent state. To implement this, a page map maintains the correspondence between the virtual pages of the database and disk blocks. AISP writes updated pages to free blocks in the log and updates the page map to reflect the new locations. When a transaction commits, the new mappings, in addition to updated pages, are written to the log. Since AISP always writes pages to free blocks, the original clustering of the blocks is lost.

In BISP the first modification to a page causes the original to be written to a free block in the log. Updates are then performed in place. The page map is used to record the locations of the shadow pages (not the original pages, since they do not move), and must be present in the log before the originals are overwritten in the database. The page map can be used to recover the last consistent state of the database. On commit, updated pages are written back to the database and the page map updated to remove the references to the corresponding shadow pages. Since BISP uses an update-in-place policy it maintains the original clustering of pages.

In the LSD updated pages are written sequentially to free blocks in the log and a page map is written to the log to record the new locations. Like AISP the original clustering of the blocks is lost. To reduce the complexity of modelling this LSD, no compaction costs are predicted. This is based on two assumptions. The first states that the disk

holding the log may be sufficiently large to avoid high degrees of fragmentation and hence avoids the need for compaction. The second states that under some workloads the extra cost of compacting the log may outweigh any benefits gained from performing a higher proportion of sequential writes. In this case the time spent performing I/O operations during compaction may be better utilised performing normal processing.

4.3.2 Categorisation of Recovery Mechanisms

In Chapter 2 each recovery mechanism is described in terms of the movement of data between a database, a cache and a log. This abstraction of each mechanism is reflected in the modelling strategy used by MaStA - each mechanism is analysed to assess its I/O costs in a number of different I/O cost categories:

- **Database reads**: The cost of data reads from the database are included in the model since the presence of a recovery mechanism may change the I/O access patterns of a running system. For this reason MaStA models total I/O costs as opposed to recovery overheads alone. For example, AISP is assumed here to incur unclustered reads.

- **Database writes**: This category includes the cost of writing non-committed data in place to the database in *undo* recovery mechanisms.

- **Log reads**: Recovery overheads such as reading page tables on restart in AISP and the LSD are included.

- **Log writes**: All data and metadata written to the log, such as writing pages of log records in a log-based system and writing updated page maps in shadow paging are calculated.

- **Propagation reads**: Recovery mechanisms that defer updates to the database may incur propagation reads. For example, an object logging mechanism must copy updated objects from the buffer to the database page containing the object. The database page must be read if it is not already in the cache.

- **Propagation writes**: These are the costs of propagating updates to the database in mechanisms that defer updates. In deferred update logging for example this consists of writing committed pages to the database during checkpoints, opportunistically or during shutdown.

- **Commit overhead**: This category includes the I/O overhead of recording the committed state of a transaction on disk. For example, this may include writing

a transaction commit record to the log in a logging system or writing the root page in a shadow paging scheme.

In the MaStA model, the four recovery mechanisms introduced in Section 4.3.1 incur costs within the I/O categories indicated in Table 4.1. Only BISP incurs database writes since the other mechanisms always write uncommitted data to the log. DataSafe incurs propagation writes to the database since it defers updates past commit time. The AISP mechanisms and the LSD incur reads on the log to recover page maps on restart.

I/O Categories		DataSafe	AISP	LSD	BISP
Data	**reads**	✔	✔	✔	✔
	writes				✔
Log	**reads**		✔	✔	
	writes	✔	✔	✔	✔
Propagation	**reads**				
	writes	✔			
Commit	**writes**	✔	✔	✔	✔

Table 4.1: Assigning I/O Cost Categories to Recovery Mechanisms

In the I/O cost predictions of recovery mechanisms made in this chapter, the cost of recovery from failure is omitted. The omission of this cost is assumed not to be significant to the overall cost of each recovery mechanism, since it is assumed that failures are infrequent, and that the overhead of providing for recovery outweighs the cost of recovering a materialised database to a consistent state.

To simplify the development of the model, the following assumptions are made:

- Main memory is large enough to hold all required page maps and data pages accessed and updated by all running transactions. This may be unrealistic in applications that execute large transactions that overflow the cache, but may be true of database applications that perform short transactions.

- All mechanisms perform the same number of database reads. The number of page faults incurred may vary marginally between mechanisms that use different page replacement algorithms but are assumed to be equal to simplify the calculation of database read costs.

4.3.3 I/O Access Patterns

The crucial contribution of the MaStA model is to distinguish various read and write access patterns, on the assumption that they may have significantly different costs. The model defines two patterns called *sequential* and *ordered*, and three patterns that are parameterised according to the degree spatial of locality. The three patterns are *clustered*, *unclustered* and *disk*. The five patterns defined are intended to reflect the characteristics of magnetic devices, but the principle applies to any device whose access time varies according to the sequence of locations accessed. The patterns are defined as follows:

- *Sequential* reads/writes (r_{seq}, w_{seq}): The data is read/written in sequentially increasing positions. This is the most efficient access pattern because hardware and software are designed to support it well. A typical example is writing to a contiguously structured log. The expectation that sequential I/O accesses exhibit good performance is based on the assumption that logically adjacent blocks are placed contiguously on physical blocks by disk controllers. Calibration measurements described in Section 4.4.1 compare the costs of *sequential* and non-sequential I/O operations, and suggest that this assumption is valid.

- *Ordered* reads/writes (r_{ord}, w_{ord}): This pattern describes I/O operations that are performed on sorted non-adjacent locations. For example, during a commit in AISP the non-adjacent blocks written may be ordered so that seek costs are minimised. The *ordered* access pattern may also encompass operations performed asynchronously, in other words, I/O requests that are scheduled in a favourable order, so if the pool of requests is sufficiently large the average cost can approach that of *sequential* I/O. A typical example is keeping a pool of committed pages requiring propagation to the database.

- *Clustered* reads/writes (r_{clu}, w_{clu}): This pattern comprises localised accesses that are synchronous and hence cannot be freely ordered. A typical example is localised database reads.

- *Unclustered* reads/writes (r_{uncl}, w_{uncl}): These are synchronous accesses within the database that involve moving the access position arbitrarily.

- *Disk* reads/writes (r_{disk}, w_{disk}): These are synchronous accesses that involve moving the access position arbitrarily far on the device. This pattern may incur higher costs than *unclustered* I/O. A typical example is forcing the log during

each commit, since the database area can be far from the log area if they are stored on the same device.

To calculate the cost of recovery mechanisms using MaStA, each I/O access pattern is assigned an average cost, which may vary between different platforms. Given a suitably accurate model of the device and associated software, one might derive an analytical or simulation model to determine the cost of each pattern. As will be seen later, the approach taken measures these values by experimentation. The refinement of I/O costs to include different access patterns turns out to be significant. For example, the ratio of the cost of the most expensive write access pattern to the least expensive is observed to be a factor of five on a particular platform.

4.3.4 Assigning I/O Access Patterns

The assignment of I/O access patterns to I/O cost categories for a given recovery mechanism is dependent on the characteristics of the mechanism. For example, a mechanism that maintains the original clustering of data performs both *clustered* and *unclustered* database reads. On the other hand mechanisms that lose the original clustering of data are assumed to always perform *unclustered* or *disk* database reads. It is conceivable that some of these mechanisms may be able to take advantage of dynamic re-clustering of data for some applications in order to perform *clustered* reads. To cater for such cases in MaStA requires only a reassignment of I/O access pattern costs to the database read categories for such mechanisms. It is assumed that application workloads have characteristics such that no effective re-clustering of pages can take place to reduce read costs.

The I/O access patterns assigned to the I/O cost categories for the four mechanisms are given in Table 4.2. In DataSafe, each database read is either *clustered* or *unclustered*. Log writes consist of writing updated pages *sequentially* to the safe, and writing pages of the safe map in an *ordered* manner to preallocated locations on disk. Committed pages are written back to the database using propagation writes. Propagation I/O can be delayed and may therefore be *ordered*. The commit I/O cost category consists of writing the root block and is assigned a *unclustered* write. Writing to the safe may also incur two *disk* seeks, if the same device is used to hold both the database and the safe: one to position the device at the safe and one to move it back to the database. The second occurs at the beginning of the next database read but is most conveniently modelled as a commit cost. Since committed changes are retained in the cache until propagated to the database, no propagation reads are required to read the changes back from the safe.

I/O Categories	DataSafe	AISP	LSD	BISP
Database Read	*clustered & unclustered*	*unclustered*	*disk*	*clustered & unclustered*
Database Write				*ordered*
Log Read		*ordered*	*ordered*	
Log Write	*sequential & ordered*	*ordered*	*sequential*	*sequential & ordered*
Propagation Read				
Propagation Write	*ordered*			
Commit	*unclustered & disk*	*unclustered & disk*	*unclustered & disk*	*unclustered & disk*

Table 4.2: I/O Access Pattern Assignments to I/O Cost Categories

In AISP, updated pages are written to free blocks. In the variation of AISP examined here, it is assumed that updated pages are written to free blocks within the database before being allocated new blocks at the end of the database. This ensures that the size of the database is minimised and so the mechanism incurs *unclustered* reads instead of *disk* reads. An alternative strategy is to extend the database when creating shadow pages and only reuse free blocks within the database when it reaches some predefined size or fills the device. This strategy would alter the characteristics of AISP to more like those of the LSD. Because the original clustering of pages is lost, database reads always require *unclustered* reads. Log reads are performed to access the page map; such reads incur *ordered* read costs. Log writes, to update the page map, can be performed in an *ordered* fashion once the device head is moved to the required location. The cost of this seek is charged to the commit I/O cost category. The commit I/O cost category also consists of writing the root block and is assigned a *unclustered* write. The additional seek incurred by the next I/O operation is also charged to the commit category.

The main difference between the LSD and AISP is that the LSD performs less expensive *sequential* log writes instead of *ordered* writes. A requirement of being able to perform *sequential* writes in the LSD is that the database is dispersed over a larger area of the device and hence database reads are assigned the more expensive *disk* read costs.

Notice that database reads in the LSD and AISP are assigned *unclustered* and *disk* read costs respectively. If the database has never been updated before and read-only applications are executed over the database, these mechanisms may be assigned the same database read patterns as DataSafe. Such workloads are not interesting in the context of this work, since they incur the same read costs under each mechanism. This thesis focuses on workloads under which there is a potential advantage in choosing

one mechanism over another. Hence in MaStA it is assumed that update queries have already been executed against the database and that the original clustering of pages of data has been lost in the LSD and in AISP.

In BISP the original clustering is maintained so database reads are either *clustered* or *unclustered.* Database writes may be performed in block order and so incur *ordered* costs. There are three costs involved in log writes. The first is writing before-images to shadow blocks in the log. Shadow blocks may be allocated contiguously and written *sequentially.* The second cost is writing the page map indicating the locations of the shadow copies. These mappings must be written before an original block is overwritten and consist of *ordered* writes. The third cost is incurred after the updated pages have been written to the database and consists of re-writing the page map to discard the locations of the corresponding shadow pages. The cost of seeking to and from the page map is charged to the commit cost category. The other commit I/O costs are as for after-image shadow paging.

4.3.5 Application Workload

The goal of the application workload abstraction is to capture the basic attributes of workloads that affect I/O. For example, the number of updates affects the number of log records or shadow pages written.

There is a trade-off between using a large number of variables to increase the expressive power of the workload abstraction and hence produce accurate I/O cost predictions, and employing fewer workload variables to ensure that the models of recovery mechanisms are tractable. The variables used (Table 4.3), are shown later to be sufficient to make qualitatively accurate comparisons of recovery mechanisms while at the same time maintaining the understandability of the analytical models of the mechanisms.

The values assigned to the workload variables may be obtained by simulation, measurement or analysis of the real application. Note that the variables used are designed to characterise workloads in page-based recovery mechanisms. Object based mechanisms may require additional variables to reflect the characteristics of workloads in terms of the objects updated and committed to the log.

Workload variables	Description
read	the number of read operations performed by the application
readRecent	the number of *read* that access data already in the cache (no page faults incurred)
readFaultLoc	the number of page faults in which the database page accessed is logically near the previously faulted page
update	the number of database updates performed by all transactions
updateTrans	the sum of the number of *update* performed by each transaction on pages already updated by the transaction
updateTemp	the number of pages updated by a transaction that have been updated by a previous transactions
updateLoc	the degree of intra-transaction update spatial locality - in the range (0,1] (affects the number of AISP and LSD page map pages updated)
firstUpdate	the number of read operations performed before the first write operation
commit	the number of update transactions committed
propWrite	the number of *update* that cause propagation writes during normal processing (in deferred update mechanisms)
propWriteFinal	the number of *update* that cause propagation writes during shutdown (in deferred update mechanisms)
db	the size of the virtual database in bytes
page	page size in bytes
mapEntry	size of a page map entry and a safe map entry in bytes
root	the number of root pages written to record a commit state in AISP, BISP and the LSD

Table 4.3: Variables Used to Characterise Workloads

4.3.6 Cost Models for the Four Recovery Mechanisms

For each I/O cost category and mechanism, workload variables are composed into workload functions to calculate the number of I/O access incurred. Table 4.4 describes the workload functions and includes their composition in terms of workload variables. The symbols ⌈ ⌉ denote the standard mathematical function 'ceiling'.

The workload functions and I/O access patterns assigned to the I/O cost categories for the mechanisms are given in Table 4.5. Within each category the cost is the product of a workload function and an I/O access pattern cost, or the sum of a number of such products. The total cost of a mechanism is the sum of the costs of its constituent I/O cost categories. As an example, when written out, the sum of the I/O cost categories for DataSafe is:

$$PMissClu \times r_{clu} + PMissUncl \times r_{uncl} + PDirty \times w_{seq} + PsafeMap \times w_{ord} + (PrWriteI + PrWriteII) \times w_{ord} + Proot \times w_{uncl} + commit \times 2 \times r_{disk}$$

The pattern r_{disk} is attributed to the commit category to indicate that seek costs are incurred by the mechanisms. Two seeks are incurred for example by DataSafe to move to the safe area and back to the database area when writing to the safe.

Workload Function	Description	Workload Variables
PMissClu	the number of *clustered* database pages faulted	*readFaultLoc*
PMissUncl	the number of non-*clustered* database pages faulted	*read - readRecent - readFaultLoc*
PDirty	the sum of the number of pages committed by each transaction	*update - updateTrans*
PTMiss	the number of page map pages read on restart in AISP and the LSD	$\frac{db / page}{page / mapEntry}$
PTDirty	the number of page map pages updated in AISP and the LSD	$commit \times \left\lceil \frac{\lceil PDirty / commit \rceil}{updateLoc \times \frac{page}{mapEntry}} \right\rceil$
PsafeMap	the number of safe map pages written to record the position of database pages in the safe	$commit \times \left\lceil \frac{\lceil PDirty / commit \rceil}{page / mapEntry} \right\rceil$
Proot	the number of root pages written in the mechanisms	*commit* × *root*
PrWriteI	the number of propagation page writes performed during normal processing	*propWrite*
PrWriteII	the number of propagation page writes performed during shutdown	*propWriteFinal*

Table 4.4: Workload Functions

	DataSafe		AISP		LSD		BISP	
I/O Category	Number of I/O	Access Pattern	Number of I/O	Access Pattern	Number of I/O	Access Pattern	Number of I/O	Access Pattern
Database Reads	*PMissClu* *PMissUncl*	r_{clu} r_{uncl}	*PMissClu* *PMissUncl*	r_{uncl} r_{uncl}	*PMissClu* *PMissUncl*	r_{disk} r_{disk}	*PMissClu* *PMissUncl*	r_{clu} r_{uncl}
Database Writes							*PDirty*	w_{ord}
Log Reads			*PTMiss*	r_{ord}	*PTMiss*	r_{ord}		
Log Writes	*PDirty* *PsafeMap*	w_{seq} w_{ord}	*PDirty* *PTDirty*	w_{ord} w_{ord}	*PDirty* *PTDirty*	w_{seq} w_{seq}	*PDirty* 2x*PTDirty*	w_{seq} w_{ord}
Propagation Writes	*PrWriteI* *PrWriteII*	w_{ord} w_{ord}						
Commit I/O	*Proot* commit × 2	w_{uncl} r_{disk}	*Proot* *commit* × 2	w_{uncl} r_{disk}	*Proot* *commit* × 2	w_{uncl} r_{disk}	*Proot* *commit* × 4	w_{uncl} r_{disk}

Table 4.5: Workload Functions and I/O Patterns Assigned to Cost Categories

4.4 Utilising MaStA

The utility and flexibility of MaStA are demonstrated by describing three applications of the model. In each, MaStA is used to predict the I/O costs of running a workload on different recovery mechanisms and different platforms. To compare the costs of a set of recovery mechanisms, for a given application and platform, three steps must be performed. These are:

1. Identify workload variables that reflect the attributes of the application's workload that affect I/O and provide values for these variables.

2. For each recovery mechanism, identify the categories in which the mechanism incurs costs and assign I/O access patterns to each category according to the properties of the I/O operations performed. For each category and recovery mechanism develop workload functions from the workload variables to calculate the number of accesses incurred of each I/O pattern.

3. Configure the model against the platform by measuring or estimating the cost of each I/O access pattern.

In each application of the model, the workload functions developed in Section 4.3 for the four recovery mechanisms (step 2) are evaluated by supplying values for the workload variables (step 1) and calibrating the I/O access patterns against two platforms (step 3).

4.4.1 I/O Access Pattern Calibration

MaStA abstracts over the characteristics of the platform by employing various I/O access patterns in the workload functions of recovery mechanisms. When utilising the models, these patterns are configured against the required platform. In the applications of MaStA described, the I/O pattern costs are obtained through measurement of the devices available on two platforms. The configurations of the platforms are:

a Sun SPARCStation ELC:

running SunOS 4.1.3,

with 48 MB main memory,

a 500 MB CDC Wren V SCSI drive dedicated to the operating system,

and a 500 MB partition on a 2.1 GB Seagate ST32151N Fast SCSI-2 (Hawk 2XL);

a DEC Alpha AXP 3000/600:

running OSF/1 V3.2,

with 128 MB main memory,

a 1 GB DIGITAL RZ26 SCSI drive dedicated to the operating system

and a 500 MB partition on a 2.1 GB Seagate ST12550N SCSI drive (Barracuda II).

The average cost of each I/O access pattern used in MaStA is measured by performing sequences of read and write operations on raw partitions. Raw partitions are used instead of files to avoid operating system disk cache effects. The sequences of I/O operations are recorded using the MaStA I/O trace format [SCM+95b] summarised in Section 5.4.4. The localities of the operations are controlled to simulate *sequential*, *ordered*, *clustered*, *unclustered* and *disk* I/O patterns. Details of the synthetic I/O traces used to measure these access patterns are included in Appendix B. Table 4.6 and Figure 4.2 give the average I/O access pattern costs measured on the SPARCStation (Hawk disk) and the Alpha (Barracuda disk), as proportions of sequential read costs.

It is important to note that these results do not compare the I/O access costs of the particular configurations of the Alpha and the SPARCStation. The results abstract over absolute costs by giving each machine's I/O access costs as multiples of the cost of a *sequential* read on that machine. ASR stands for Alpha Sequential Read and SSR for SPARCStation Sequential Read.

I/O Access Pattern	Alpha	SPARCStation
Sequential reads (r_{seq})	1.0 ASR	1.0 SSR
Sequential writes (w_{seq})	1.6 ASR	1.0 SSR
Ordered reads (r_{ord})	3.8 ASR	2.7 SSR
Ordered writes (w_{ord})	2.4 ASR	2.6 SSR
Clustered reads (r_{clu})	3.8 ASR	4.0 SSR
Clustered writes (w_{clu})	3.1 ASR	3.8 SSR
Unclustered reads (r_{uncl})	4.3 ASR	4.6 SSR
Unclustered writes (w_{uncl})	3.7 ASR	4.6 SSR
Disk reads (r_{disk})	4.9 ASR	4.9 SSR
Disk writes (w_{disk})	4.4 ASR	5.2 SSR

Table 4.6: Average Costs of I/O Access Patterns

The results highlight two issues fundamental to the manner in which MaStA distinguishes between I/O access patterns. The first is that the relative costs of different I/O patterns vary significantly. For example, the ratio of the cost of

sequential reads to *disk* writes is a factor of 5 on the SPARCStation. The second is that the relative cost of I/O access patterns may vary across different platforms. For example, the ratios of *sequential* write costs to *disk* write costs on the given Alpha and SPARCStation configurations are 1:2.7 and 1:5.2 respectively.

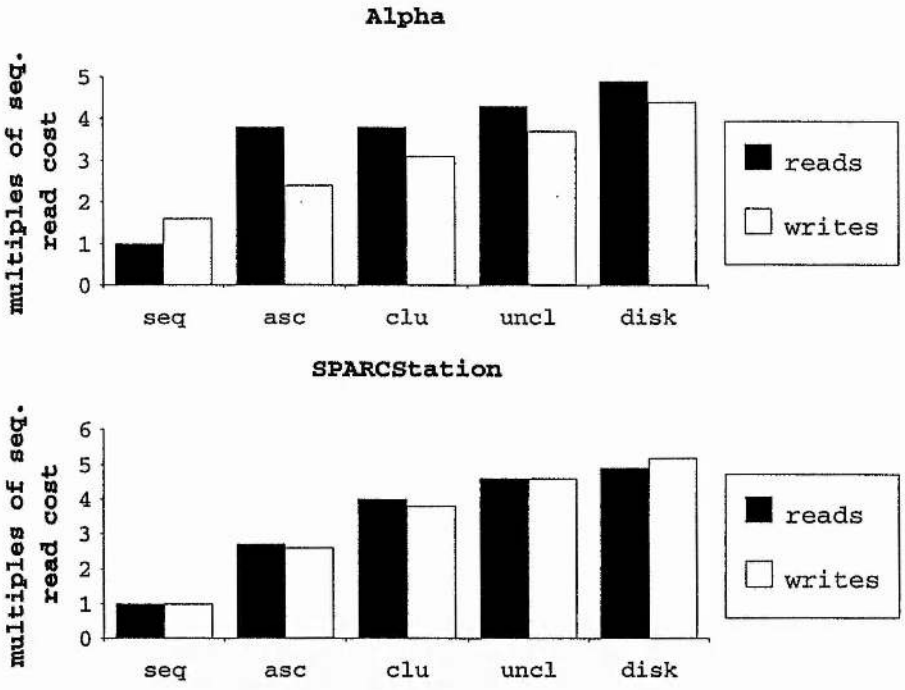


Figure 4.2: Average Costs of I/O Access Patterns

4.4.2 Applications of the Model

The following applications of MaStA demonstrate how the workload functions developed for the four recovery mechanisms may be used to predict costs under various workloads on the given SPARCStation and the Alpha configurations. Each application defines a workload and varies one or more of the workload variables to illustrate the sensitivity of the model to those variables. The workload functions are evaluated using the I/O access pattern costs recorded in Table 4.6. In addition, the functions are evaluated using a uniform I/O cost to emphasise the need to differentiate I/O access pattern costs.

4.4.2.1 Application 1

Application 1 considers the relative costs of recovery mechanisms under workloads with varying degrees of update frequency. The workload variable values are given in Table 4.7. The value of *update* is varied while the number of read operations remains constant. The value of *updateTrans*, *propWrite* and *propWriteFinal* are varied in

proportion to *update* to ensure that the ratio of propagation and log writes to updates remains constant.

Workload Variables	Values
read	1000000 pages
readRecent	800000 pages
readFaultLoc	100000 pages
update	0→400000 pages
updateTrans	$3/4 \times$ *update* pages
updateLoc	20%
commit	500
propWrite	$1/20 \times$ *update* pages
propWriteFinal	$1/100 \times$ *update* pages
page	8192 bytes
mapEntry	8 bytes
root	1 page
db	120 MB

Table 4.7: Workload Variable Values in Application 1

The graphs in Figure 4.3 illustrate the results of evaluating the workload functions developed for the four recovery mechanisms. The three graphs correspond to three sets of values assigned to the I/O access patterns: the SPARCStation's, the Alpha's and a uniform set where each pattern is given the same cost. For each set of access pattern values, the predicted costs incurred by each recovery mechanism are shown as multiples of the *sequential* read cost in the set.

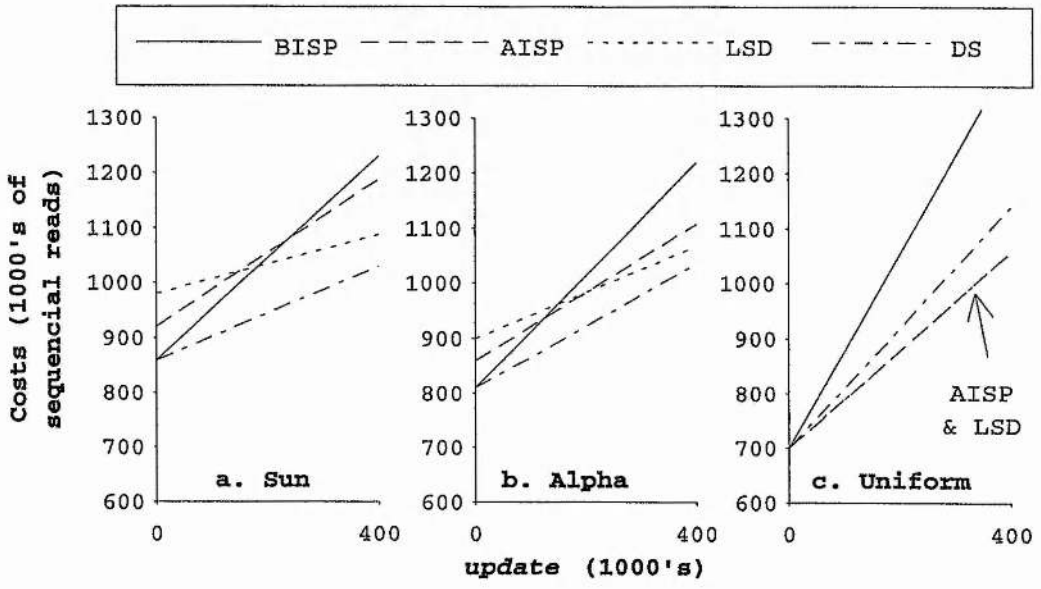


Figure 4.3: Results from Application 1

Figures 4.3.a and 4.3.b show that when the update frequency is low the LSD and AISP incur higher costs than DataSafe and BISP. This is because LSD and AISP perform only *disk* and *clustered* database reads respectively, whereas BISP and DataSafe incur some lower costing *clustered* reads as well as *unclustered* reads.

As expected the I/O costs of all the mechanisms increase as the frequency of updates increases. The graphs illustrate that the cost of BISP increases more rapidly compared to the other mechanisms. This is because committing a page causes two writes: the first, to write the before-image of the page to the log and the second to write the updated page to the database. In DataSafe, a page may be updated and committed to the safe a number of times for each time it is written to the database, hence the rate of increase of its costs is lower.

Figure 4.3.c illustrates the relative costs of the mechanisms calculated using a uniform cost for each I/O pattern. As can be seen the relative positions of the costs of the recovery mechanisms in Figure 4.3.a and 4.3.b vary, depending on the particular workload, while they do not in Figure 4.3.c. This is because the cost of each mechanism in a uniform model is based only on the number of I/O operations performed, whereas the relative costs of mechanisms also depend on the variations in the costs of the different access patterns performed. This is also why the costs of AISP and the LSD are equal when their workload functions are evaluated using a uniform I/O cost (in Applications 1, 2 and 3).

4.4.2.2 Application 2

Application 2 illustrates the effects on I/O costs of varying spatial locality of read faults. The workload variable values are given in Table 4.8. The locality of read faults is varied by changing *readFaultLoc* between 10000 (poor locality) and 190000 (good locality). This means that each workload performs 200000 read faults (*read-readRecent*), but the workloads vary, in that they perform between 10000 and 190000 localised read faults.

At the left hand side of each graph in Figure 4.4, workloads perform mainly unclustered reads on the database, and the right hand side represents workloads that perform mainly localised reads. Figure 4.4.a and 4.4.b illustrate that as read locality increases, the costs of BISP and DataSafe reduce. This is because an increasing proportion of database reads incur *clustered* costs in these mechanisms. On the other hand AISP and the LSD incur only *unclustered* and *disk* database reads costs respectively for all workloads and hence their costs do not vary with changes in read locality.

Workload Variables	Values
read	1000000 pages
readRecent	800000 pages
readFaultLoc	10000→190000 pages
update	100000 pages
updateTrans	80000 pages
updateLoc	20%
commit	500
propWrite	15000 pages
propWriteFinal	2000 pages
page	8192 bytes
mapEntry	8 bytes
root	1 page
db	120 MB

Table 4.8: Workload Variable Values in Application 2

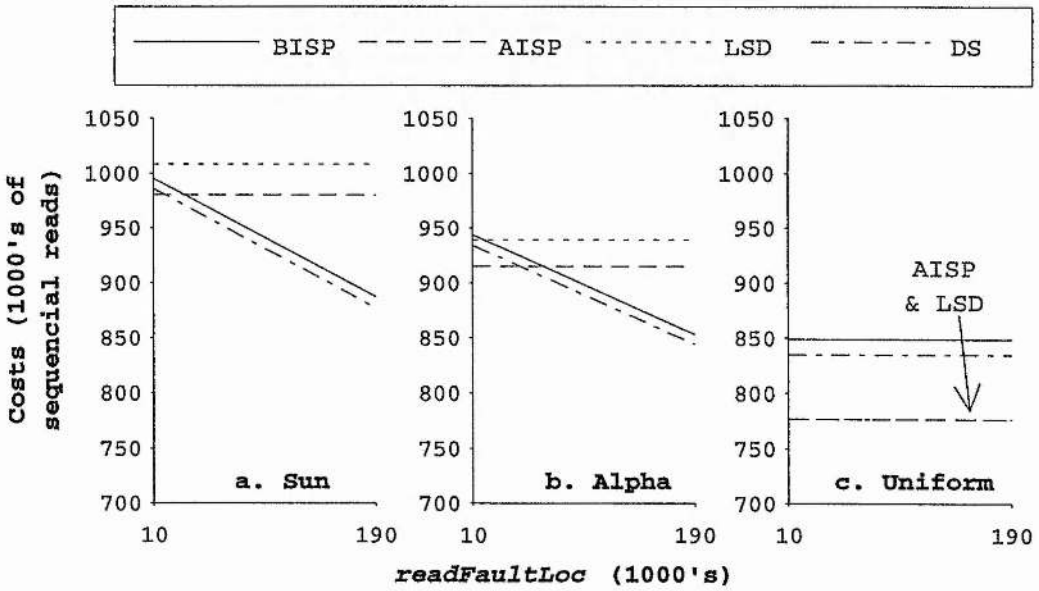


Figure 4.4: Results from Application 2

Notice that under workloads with poor locality the cost of AISP is lower than the cost of DataSafe and BISP. This is because under these workloads, all mechanisms perform non-*clustered* reads and because AISP incurs lower write costs since it only performs a single write for each page committed. The LSD incurs higher costs than AISP due to the more expensive *disk* read costs that the LSD performs.

No variation is seen using a uniform I/O cost (Figure 4.4.c) since these costs are based only the number of I/O operations performed and do not take account of the difference between the costs of *clustered*, *unclustered* and *disk* reads.

4.4.2.3 Application 3

Application 3 illustrates the effects on the costs of DataSafe of varying temporal locality of updates. The degree of update temporal locality is varied by changing the value of *propWrite*, i.e. by varying the proportion of updates that cause propagation writes The other variables remain constant (Table 4.9). The poorest locality is achieved by setting *propWrite* to 99800. This means that of the 100000 pages committed (*PDirty*), 99800 are written to safe and propagated to the database before being updated again. The remaining 200 propagation writes are attributed to *propWriteFinal*. This scenario represents an application that performs sparse updates on a large database using a small cache. Maximum locality is achieved by setting *propWrite* to 0, i.e. no pages are propagated to the database during normal processing). In other words, on average each transaction updates and commits the same 200 (*PDirty/commit*) pages.

Workload Variables	Values
read	1000000 pages
readRecent	800000 pages
readFaultLoc	190000 pages
update	200000 pages
updateTrans	100000 pages
updateLoc	20%
commit	500
propWrite	99000→0 pages
propWriteFinal	200 pages
page	8192 bytes
mapEntry	8 bytes
root	1 page
db	120 MB

Table 4.9: Workload Variable Values in Application 3

The left hand side of each graph (Figure 4.5) represents workloads in which transactions update pages that have not recently updated (low temporal locality of updates). At the right hand side, each transaction updates the pages changed by a recent transaction. As expected the cost of DataSafe reduces as the degree of update locality increases, due to the reduction in the number of propagation writes. The costs of the other mechanisms do not vary because each page that is committed causes a fixed number of writes.

Under this workload AISP incurs higher costs than BISP on the SPARCStation configuration (Figure 4.5.a), and vice versa on the Alpha configuration (Figure 4.5.b).

This is mainly because the cost of *sequential* writes, to record shadow pages in BISP, are more expensive relative to other I/O patterns on the Alpha than on the SPARCStation. This result highlights that variations in the relative costs of I/O patterns across different platforms may be sufficient to cause the ordering of the costs between mechanisms under a particular application to differ on the platforms.

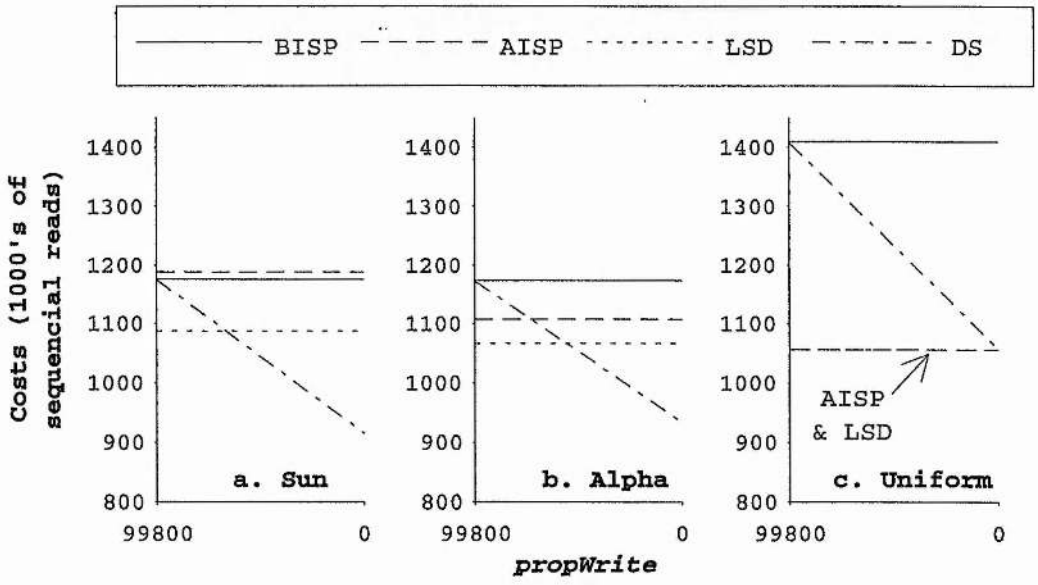


Figure 4.5: Results from Application 3

4.5 Conclusions

Chapter 1 introduced a flexible database architecture that may be configured to provide optimum performance for a particular application. To effectively configure recovery management in such an architecture, the recovery scheme that incurs the lowest cost for the application must be selected. To meet this demand, a new analytical I/O cost model called MaStA is introduced. The model reduces the complexity of predicting the costs of recovery mechanisms by categorising I/O operations in terms of the movement of data between a database, a log and a cache. The number of I/O operations incurred in each category is estimated using a workload abstraction that takes into account variables that affect I/O. To accurately calculate the cost of each category, MaStA differentiates I/O access patterns, such as *sequential* and *unclustered*, the costs of which may be calibrated against the platform being used to run the application.

Applications of the model demonstrate the flexibility and utility of MaStA. The applications involve calibrating the I/O patterns against two platforms and providing values for the workload abstraction with which to evaluate the workload functions

developed for four recovery mechanisms. Comparisons of the resulting I/O cost predictions highlight a number of issues:

- The variations between the costs of different I/O access patterns affect total costs significantly. Furthermore the I/O costs of mechanisms that perform the same number of I/O accesses can only be differentiated if different I/O patterns are modelled.

- The relative costs of mechanisms may vary on different platforms under the same application, hence an analytical model should allow each I/O access pattern to be calibrated against the particular platform on which the application is executed.

- The relative costs of recovery mechanisms are highly dependent on workload characteristics. In particular, the results corroborate the belief that no one mechanism incus the lowest cost for all applications.

The MaStA model is used in Chapter 7 in a worked example of the flexible Flask architecture, to choose the appropriate recovery mechanism for a particular application and platform.

A number of assumptions are made in the development of MaStA. These are discussed in the next chapter, along with a framework designed to validate the assumptions.

5 Validation Strategy of MaStA

5.1 Introduction

MaStA is an analytical I/O cost model that estimates performance for a particular combination of application workload, recovery mechanism and execution platform at relatively low cost. To recap, the main features of the model are:

- Cost is based upon a statistical estimation of disk activity, broken down into I/O cost categories for each recovery mechanism.
- The model may be calibrated with different disk performance characteristics, either simulated, measured by experiment or predicted by analysis.
- The model is usable over a wide variety of workloads, including those typical of object-oriented and database programming systems.

This chapter introduces the four underlying assumptions of the MaStA model and presents the validation framework designed to verify the assumptions. The procedures performed to validate the assumptions, and the corresponding results are discussed in Chapter 6.

5.2 Assumptions

Three major abstractions are made to describe MaStA, based on critical underlying assumptions. The abstractions are:

- the recovery mechanism abstraction;
- the disk performance abstraction;
- and the workload abstraction.

5.2.1 Recovery Mechanism Abstraction

The recovery mechanism abstraction assigns I/O cost categories to each recovery scheme. The total cost derived by the model is the sum of these categories. The purpose of the categorisation is to reduce the complexity of comparing recovery mechanisms and improve the analysis of the mechanisms. The success of this abstraction depends heavily upon two assumptions:

I/O Assumption: In applications where variations in total costs of using different recovery mechanisms are significant, the variations in the CPU costs incurred are insignificant compared to the variations in the I/O costs.

Cost Category Interaction Assumption: The interaction between the different categories of I/O accesses is not significant; that is, the cost of running the I/O stream generated by a given recovery mechanism is not significantly different from the sum of the costs of running the streams of each I/O cost category separately.

5.2.2 Disk Performance Abstraction

MaStA abstracts over the characteristics of the device by employing various I/O access patterns in the models of recovery mechanisms. The average cost of each pattern may be obtained either by simulation, experiment or by further analysis of the device in question. This abstraction depends on the assumption:

Access Pattern Cost Assumption: To make predictions of the relative costs of recovery mechanisms for all workloads, it is sufficient to assign a predicted average cost to each I/O access pattern.

5.2.3 Workload Abstraction

The last abstraction in MaStA is over the workload associated with the application. As the interest is only in I/O behaviour, this need not encompass any CPU activity of the application, but only its data accesses. The application is characterised in terms of workload variables such as the number of database read operations, read locality and update frequency.

Workload Assumption: The cost of running the I/O stream generated by an application is approximately the same as running the I/O stream generated by the workload abstraction.

5.3 Overview of the Validation Strategy

The strategy used to validate the assumptions of MaStA [SCM+95a, MCM+95] is outlined in Figure 5.1. A variety of workload traces produced by a synthetic workload generator and by the OO1 and OO7 benchmarks are recorded. The OO1 and OO7 benchmarks are widely accepted as a basis on which different object oriented database systems may be compared and are used here (implemented in Napier88 [MBC+89]) to provide typical database workloads. Each workload trace records the database accesses performed by a particular benchmark query and allows the same workload to be executed multiple times on different recovery mechanisms and platforms.

The workload traces are executed on three recovery mechanisms: AISP, DataSafe and the LSD developed in Chapter 3, and on two platforms: a Sun SPARCStation and a

DEC Alpha configured with different devices and operating systems. A summary of the configurations of the recovery mechanisms used is included in Appendix A.1.

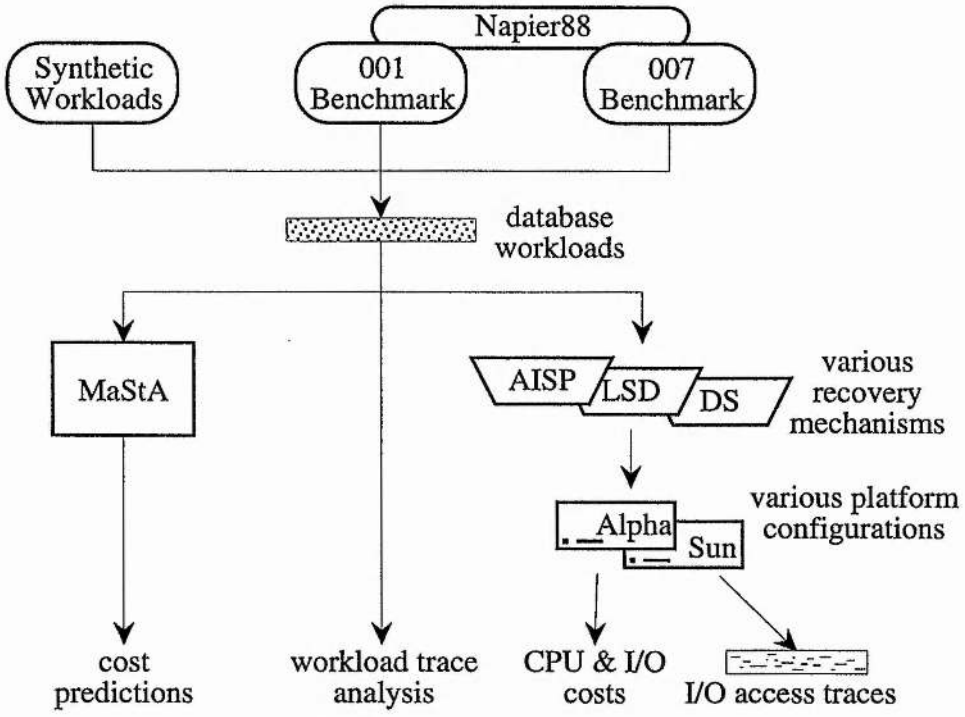


Figure 5.1: The MaStA Validation Strategy

The I/O and CPU costs of executing each workload trace are measured and traces of the I/O accesses performed are recorded. The workload traces are characterised in MaStA to provide I/O cost predictions of the workloads. The predicted and real I/O costs, the I/O traces and the database workload traces are analysed in Chapter 6 to validate the assumptions that support the abstractions of MaStA. A strength of this strategy is that by validating each assumption for more than one platform, operating system and device, it illustrates the independence of the MaStA assumptions from these components. Section 5.4 provides details of the components of the validation strategy.

Once the MaStA model has been validated there is a final assumption that is used in estimating the cost of any combination of application, mechanism and platform. The assumption is that there are no significant phase changes in the performance of recovery mechanisms [ABJ+92]. In other words, small changes in workload or platform characteristics do not cause dramatic changes in the relative costs of recovery mechanisms.

5.4 Validation Framework Design

A number of components are common to the procedures performed to validate the assumptions of MaStA. These are:

- the persistent system employed to generate database workloads traces and the format of the workload traces;
- the benchmarks used to generate workloads typical of database applications;
- the platforms used to execute the workloads;
- and the format of the I/O traces.

5.4.1 Napier88 and Workload Traces

The Napier88 system [MBC+89] is employed to generate the traces used in the validation strategy. The Napier88 compiler maps programs onto an interpreted abstract machine, the Persistent Abstract Machine [CBC+89] which accesses persistent data through a persistent heap interface. The persistent heap in turn accesses data on non-volatile storage through a recovery manager.

The validation strategy records traces of I/O accesses and traces of database workloads, and analyses the traces off-line to avoid potential sources of interference. Each database workload trace records the database read, write and commit operations performed by a particular benchmark query. Read and write trace entries record the length of data accessed and the database addresses of the data. The I/O access traces are obtained by modifying each recovery mechanism to record the I/O operations performed. The format of I/O access traces is discussed in Section 5.4.4.

An advantage of using Napier88 is that the source code of the system is available allowing the required database and I/O accesses traces to be recorded. It is not possible to record these traces from many commercially available databases due to the unavailability of the source code. An alternative method that could have been used to record I/O traces would have been to use commercially available database systems executing on an operating system such as LINUX for which the source is available. Access to the operating system's source code would allow the device drivers to be augmented to record traces of the I/O accesses performed by the database systems.

An assumption of recording I/O access traces and database workload traces is that recording traces does not significantly affect the execution of the system. This assumption is validated by performing an experiment that compares the total elapsed

times of executing Napier88 applications while recording I/O and workload traces against the elapsed times when no traces are recorded. The results of performing this experiment indicate that recording traces has no significant effect on the results of the validation procedures.

Ideally more than one database system should be used to generate workloads and I/O traces in the validation framework. Using only one system may be justified by the fact that database workload traces taken from Napier88 are executed instead of using the system directly. Therefore interference from the Napier88 interpreter is factored out. Future work shall investigate the inclusion of other database systems in the validation framework.

5.4.2 Benchmarks

Validation of the MaStA assumptions for all possible workloads is approximated in the strategy by employing four benchmarks to generate workloads typical of database systems. The particular configurations of the benchmarks described provide fixed workloads for which each assumption may be validated. To promote confidence that the results are independent of the configuration of the benchmarks two configurations of OO1 are used. This section provides a summary of the benchmarks. Details of the configuration of each benchmark are included in Appendix A.2.

5.4.2.1 OO1

The OO1 benchmark [CS92] (introduced in Section 2.5.3.1) generates workloads that are supposed to be typically found in engineering applications such as CAD/CAM. The benchmark provides three standard queries called `lookup`, `traverse` and `insert` which execute against a database containing interconnected parts.

`lookup`: A set of random part identifiers is first generated. Read-only transactions are then executed, each of which fetches the set of parts from the database.

`traverse`: A set of read-only transactions are executed. Each transaction selects a part at random and recursively traverses the connected parts to a specified depth. A null procedure is called for each part traversed.

`insert`: A set of transactions is executed, each of which inserts new parts and commits. Each part inserted is connected to a number of other (randomly selected) parts.

In the validation strategy, three additional queries are executed against the OO1 database to provide a wider range of workloads. They are:

`scan`: This read-only query is included to provide a workload typical of applications that perform linear scans of data, such as a database application that scans all customer records to gather some statistics.

`insertLarge`: In an attempt to highlight any effects on the validation results of varying workload sizes, this additional query is included to generate a larger workload to that of the standard `insert` query. In particular, `insertLarge` executes more transactions than `insert` and each transaction inserts a larger number of new parts.

`update`: Since the standard OO1 benchmark provides only insert or read-only queries this query is included to provide a workload that updates existing data in the database.

Some recovery mechanisms, such as AISP and the LSD, perform dynamic reclustering of data in the database during update transactions. To highlight any effects on subsequent reads of performing reclustering, an additional set of read-only queries is executed after the update queries (`insert`, `insertLarge` and `update`). The second set of read-only queries generate similar workloads to those of the first set (`lookup`, `scan` and `traverse`). The read-only queries in the second set are called `lookup2`, `scan2` and `traverse2` to differentiate the results of executing the two read-only sets.

5.4.2.2 OO1b

The second benchmark is the OO1 benchmark configured with a larger database and larger workloads. This configuration of OO1, called OO1b in the validation strategy, is used in an attempt to show that the validation results are independent of the configuration of the benchmarks used. A number of alterations are made to the queries to increase workload sizes in OO1b (see Appendix A.2 for details). These are:

`lookup`, `lookup2`: The set of random part identifiers generated, and accessed by each transaction, is enlarged.

`insert`, `insertLarge`: The number of new parts entered into the database by each transaction is increased.

`update`: The number of parts read and updated by each transaction is increased.

5.4.2.3 OO7

The third benchmark is OO7 [CDN93] (introduced in Section 2.5.3.2). It is designed to provide performance metrics for comparing various components of OODBMSs. A drawback of the OO7 benchmark with regard to the validation strategy is that through experimentation it was found that Napier88 tended to translate the complex OO7 queries into CPU bound workloads. The validation strategy however is concerned with validating the assumptions of MaStA - a model designed to predict costs of I/O bound workloads. Hence, only a representative cross-section of the three categories of OO7 queries (traversals, queries and structural modifications) are included in the validation strategy. The queries used are:

`T1`: The OO7 database is traversed, visiting the unshared composite parts of each base assembly visited. As each composite part is visited, a depth-first traversal is carried out on its subgraph of atomic parts.

`T6`: Traversal `T1` is repeated, visiting only the root part of each composite part.

`Q2`: A range of build dates which contains the last 1% of the dates found in the database's atomic parts is chosen and these parts are retrieved.

`Q8`: All pairs of documents and atomic parts with matching identifiers are found.

`S2`: The most recently created composite parts are removed in their entirety, including document objects and atomic part subgraphs.

5.4.2.4 MaStA Object Benchmark

The OO1 and OO7 benchmarks are designed to allow both CPU and I/O costs of database system components to be analysed. In some workloads such as those generated by OO7 the I/O costs can be insignificant. To provide workloads that incur high proportions of I/O costs, an I/O bound benchmark called MOB (MaStA Object Benchmark) is also included in the validation strategy. This benchmark consists of a database of large objects indexed by a B+tree, and a number of queries. The queries are designed to exhibit various locality properties and vary in the number of objects accessed and updated.

`scan`: All objects in the database are read once in index order.

`readTrans`: A set of read-only transactions are executed. Each transaction reads objects chosen at random from a contiguous range of 10% of the

database. The index of the first object in each range is chosen at random for each transaction.

`randomAcc`: A set of objects chosen at random are accessed.

`updateTrans`: A series of update transactions are executed. Each transaction reads and updates objects chosen using the selection algorithm used in `readTrans` and commits.

`RWtrans`: A set of update transactions are executed. Each transaction reads objects chosen using the selection algorithm used in `readTrans`, updates the last object accessed and commits.

`randRWtrans`: A set of update transactions are executed. Each transaction reads objects chosen at random, updates the last object accessed and commits.

`scan2`, `readTrans2`, `randomAcc2`: These queries generate similar workloads to those of `scan`, `readTrans` and `randomAcc` respectively. Similarly to the OO1 benchmarks they are included in the benchmark to highlight any residual effects of providing recovery during the three update queries.

5.4.3 Platforms

A strength of the validation strategy is that each assumption of MaStA is verified for a number of different platforms, operating systems and devices, and hence the validation results are less likely to depend on the particular attributes of any one of these components. The platform configurations used in the framework are:

- a Sun SPARCStation ELC:

  running SunOS 4.1.3,

  with 48 MB main memory,

  a 500 MB CDC Wren V SCSI drive dedicated to the operating system,

  and a 500 MB partition on a 2.1 GB Seagate ST32151N Fast SCSI-2 (Hawk 2XL);

- a DEC Alpha AXP 3000/600:

  running OSF/1 V3.2,

  with 128 MB main memory,

a 1 GB DIGITAL RZ26 SCSI drive dedicated to the operating system

and a 500 MB partition on a 2.1 GB Seagate ST12550N SCSI drive (Barracuda II).

- memory usage: 8 MB dedicated to the recovery mechanism's cache and the remainder for the process running the database workloads and the operating system.

5.4.4 I/O Trace Format

To analyse the I/O operations performed by recovery mechanisms the operations are recorded in traces using the MaStA I/O trace format described in [SCM+95b]. The aim of this format is to standardise the manner in which the I/O operations performed by database systems are recorded and to allow trace consumers to develop analysis tools that operate on such traces. The trace format is designed to be independent of machine architecture by defining the byte ordering of trace entries and enabling the configuration of the platform such as the disks used, to be recorded. It is also independent of the recovery mechanism used by allowing the configuration of mechanisms to be recorded.

Each I/O trace is composed of a sequence of entries each of which records a read, write or synchronisation operation performed by a recovery mechanism executing a particular application. Reads and writes are recorded in a trace as operating on one or more blocks. Each read or write also operates on a particular logical area of storage such as the database or the log. Synchronisation operations operate on one or more areas. The validation strategy requires that each I/O operation performed by database systems can be associated with the MaStA I/O cost category (database read, propagation write, etc.) in which it is performed. The trace format allows an I/O cost category to be recorded with each I/O operation.

Configuration entries may be included in a trace to record additional information to be used by trace consumers. For example, each logical area of storage used by a mechanism is distinguished from others using a region entry. Each region entry records which device an area is bound to and the location on the device of the beginning of the area. Configuration entries may also record additional information such as text describing the platform.

5.5 Conclusions

Three major abstractions are made to simplify the development of the MaStA cost model: recovery mechanism, disk performance and workload. These abstractions are based on four assumptions.

- In applications where variations in total costs of using different recovery mechanisms are significant, the variations in the CPU costs incurred are insignificant compared to the variations in the I/O costs.
- The interaction between the different categories of I/O accesses is not significant; that is, the cost of running the I/O stream generated by a given recovery mechanism is not significantly different from the sum of the costs of running the streams of each I/O cost category separately.
- To make predictions of the relative costs of recovery mechanisms for all workloads, it is sufficient to assign a predicted average cost to each I/O access pattern.
- The cost of running the I/O stream generated by an application is approximately the same as running the I/O stream generated by the workload abstraction.

This chapter has described the framework employed to validate these assumptions to gain confidence that MaStA can be used to make accurate comparisons of recovery mechanisms. The strategy involves executing benchmarks designed to generate workloads typical of database applications. The workloads are executed on three recovery mechanisms: AISP, DataSafe and a LSD, and on two platforms configured with different devices and operating systems. During each execution the I/O and CPU costs are measured and traces of I/O accesses are recorded. These costs, the costs predicted using MaStA, the I/O traces and the database workload traces are analysed in Chapter 6 to validate the MaStA assumptions. A strength of this strategy is that by validating each assumption for multiple combinations of recovery mechanism, platform, operating system and device, it illustrates the independence of the MaStA assumptions from these components.

6 Validation Procedures

6.1 Introduction

The previous chapter introduced the assumptions that underly the MaStA I/O cost model and discussed the design of the framework employed to validate these assumptions. This chapter breaks the framework down into four procedures each of which is composed of a number of experiments designed to verify one of the assumptions. The results of the experiments performed are analysed to determine whether the specific assumption is valid. A total of 2268 experiments were performed in the validation strategy.

A number of strategies are used in the framework to avoid interference from the operating system. These strategies are discussed followed by a description of each validation procedure and the corresponding results. Having validated the four assumptions of MaStA, the model is used to predict the I/O costs of recovery mechanisms executing the workloads used in the validation framework. The accuracy of these cost predictions is verified by comparing them against empirical measurements of the workloads.

6.2 Avoiding Interference

6.2.1 Platform Interference

File systems commonly make use of optimisations such as caching, prefetching and re-ordering of I/O operations to reduce I/O costs. The use of raw partitions instead of file systems avoids these optimisations and increases the probability that I/O operations are carried out at the disk level at the time and in the order they are performed by the application - a requirement in Section 6.4. For example, it is important that synchronous *unclustered* I/O operations are performed synchronously at the disk level. If a file system had been used it may have cached and re-ordered the I/O operations in an attempt to reduce costs and hence may have affected the validation results.

It may be possible for MaStA to make cost predictions of recovery mechanisms running on file systems instead of raw partitions. However, without knowledge of the behaviour of the file system, such as the caching policy used, it may be difficult to obtain accurate results. Experience with running recovery mechanisms on a file system and over a raw partition has shown that optimisations incorporated into the system can cause the I/O throughput of the file system to be lower than that of the raw partition. For example, on a particular platform it was found that in many workloads

that updated a database, the operating system swapped out the virtual address space of the application in order to cache a write-only log file. To use MaStA to accurately compare the costs of recovery mechanisms on file systems, such interference must first be removed.

Another potential source of interference comes from using modern disk controllers that can re-order and cache I/O operations. To an extent, the disk abstraction of the MaStA model takes such optimisations into account by calibrating each I/O access pattern used in the model against the device. So, for example, if a disk controller optimises *clustered* write operations, the model reflects the optimisation in I/O cost predictions by calibrating the access pattern against the disk (see Section 4.4.1 on calibrating I/O access patterns).

Since no operating system caches are used, each validation experiment is effectively performed using a cold system thus avoiding the requirement to flush caches between each experiment.

Disk performance can vary across different areas of a device, for example, due to variations in data density. To ensure that the results of the experiments performed in each validation procedure are comparable, the same area of disk is used for each experiment. A more comprehensive strategy would involve performing each validation procedure over a number of different areas of the device.

In each validation procedure, the platforms are run in single user mode to reduce network interference, interference from other processes and the operating system.

6.2.2 Experimental Interference

To investigate whether the I/O costs measured in the validation procedures are accurate, the costs are recorded using two methods (Figure 6.1). The first measures the cost of individual I/O operations using the standard library functions `gettimeofday` and `getrusage`. The function `gettimeofday` records the elapsed time of each I/O function call and `getrusage` is used to factor out the CPU cost incurred during each call. The second measurement calculates I/O costs by subtracting the total CPU costs, and the I/O cost of reading database workload traces and I/O traces, from the total elapsed time recorded using the `time` command provided by SunOS and OSF. An average variation of 1.8% was observed between the two methods of measuring I/O costs.

Each experiment in the validation procedures is performed a number of times so that any fluctuations in the costs measured may be factored out. From the results obtained,

it was found that three executions of each experiment were sufficient to obtain consistent results. In particular, the average variation in the costs of the three executions of any given experiment was less than 1.5% of the average cost of the experiment.

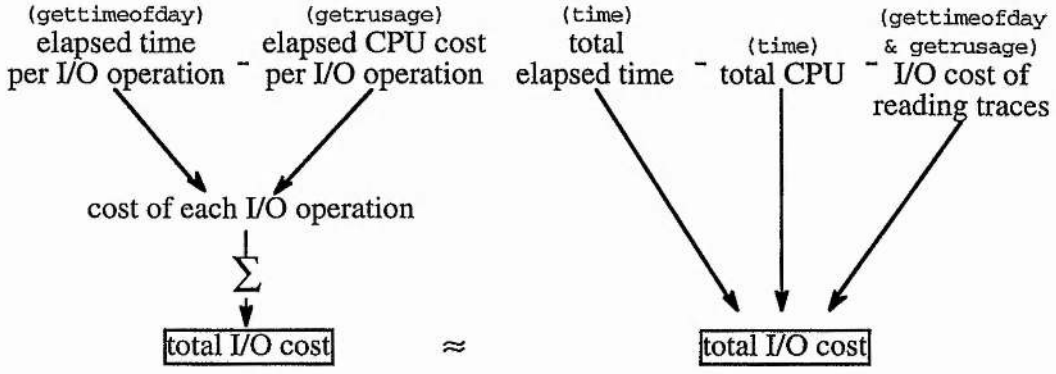


Figure 6.1: The Two Measurements Taken in the Validation Procedures

6.3 Validation of the I/O Assumption

The requirement of this procedure is the justification of the hypothesis (the I/O Assumption of MaStA):

> In applications where variations in total costs of using different recovery mechanisms are significant, the variations in the CPU costs incurred are insignificant compared to the variations in the I/O costs.

The workloads generated from OO1, OO1b, MOB and OO7 (discussed in Section 5.4.2) are executed on AISP, the LSD and DataSafe (DS). The I/O and CPU costs of running each workload are measured (Figure 6.2).

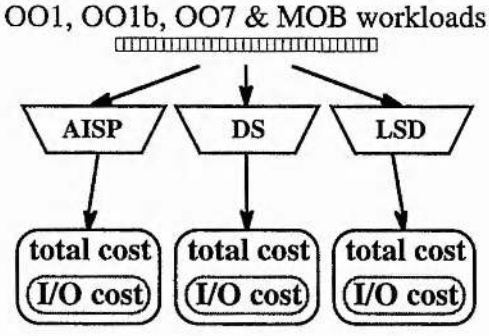


Figure 6.2: The Costs Measured to Validate the I/O Assumption

The hypothesis is justified if for each pair of recovery mechanisms, where the variation in the total costs of executing a given workload is significant, the I/O costs can be used to predict which mechanism incurs the lower total cost. The variation in

the total costs of two mechanisms is considered significant if the variation is greater than 5% of the lower total cost. There are 103 such variations in the workloads executed.

6.3.1 Results

The average I/O cost and total cost of each workload executing on each of the three recovery mechanisms and on the two platforms are given in Appendix C.1. Analysis indicates that in all 103 comparisons of recovery mechanisms where there is a significant total cost variation, the I/O costs could be used to predict which mechanism incurs the lower total cost. For example, if the I/O costs of DataSafe and the LSD executing `insert` (OO1b) on the configuration of the Sun are compared (68.20 and 84.93 seconds respectively) then DataSafe is predicted to incur the lower total cost. This is verified by comparing the total costs of the mechanisms (76.63 and 95.07 seconds respectively). Further analysis indicates that the I/O costs can also be used to predict which of a pair recovery mechanism incurs the lower total cost for a given workload when the total cost variation between the mechanisms is between 1% and 5%.

The results verify that for the workloads which exhibited significant total cost variations, the differences in CPU costs are insignificant when the relative total costs of recovery mechanisms are being compared. The justification of this hypothesis suggests that MaStA needs only predict the I/O costs of recovery schemes to compare their relative performances for a given application.

6.4 Validation of the Cost Category Interaction Assumption

The requirement of this procedure is the justification of the hypothesis (the Cost Category Interaction Assumption of MaStA):

> The interaction between the different categories of I/O accesses is not significant; that is, the cost of running the I/O stream generated by a given recovery mechanism is not significantly different from the sum of the costs of running the streams of each I/O cost category separately.

The workloads generated from OO1, OO1b, MOB and OO7 are executed on AISP, the LSD and DataSafe, recording traces of the I/O operations performed (Figure 6.3). Each I/O trace is then ordered by MaStA I/O cost category (database reads, log writes, etc.) to produce a set of ordered traces. The original traces and the ordered traces are then run on the raw partitions to measure the I/O costs. The hypothesis is justified if the relative costs of running the ordered traces generated from any two recovery

mechanisms for a given workload reflect the relative costs of running the corresponding original I/O traces. In other words, if the cost of the original I/O trace generated from a mechanism is less than the cost of the original I/O trace generated from another mechanism, then this should also be true of the ordered I/O traces generated from the two mechanisms.

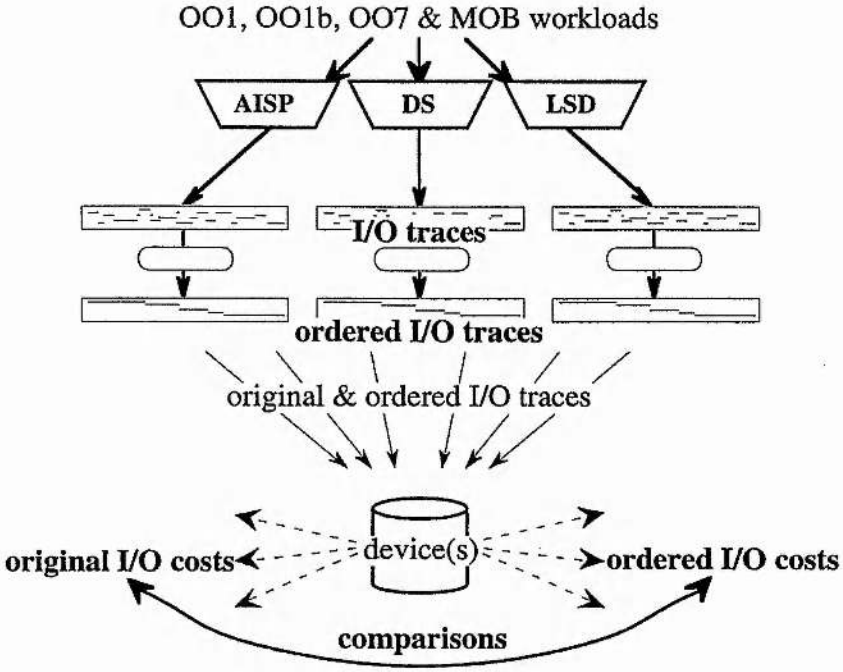


Figure 6.3: Costs Measured to Validate the Cost Category Interaction Assumption

6.4.1 Results

The average costs of running the original and the ordered I/O traces on the configurations of the Sun and the Alpha are given in Appendix C.2. As an example, the graphs in Figure 6.4 illustrate the costs of running the original and ordered I/O traces of MOB on the configuration of the Sun. Analysis of all the results in Appendix C.2 indicates that the average variation between the cost of running an ordered I/O trace and the cost of running the corresponding original I/O trace is 2.1%. The largest cost variation observed in all the measurements recorded, is on the configuration of the Sun running MOB on DataSafe - the cost of running the original trace generated by the `updateTrans` query is 15.3% lower than the corresponding ordered I/O trace (DS graph in Figure 6.4). The variation may be caused by seeks incurred in the original trace to move the device head to the end of the log when transactions commit, in contrast to the ordered trace, where overall I/O costs are reduced since seeks to the end of the log are avoided during the log writes.

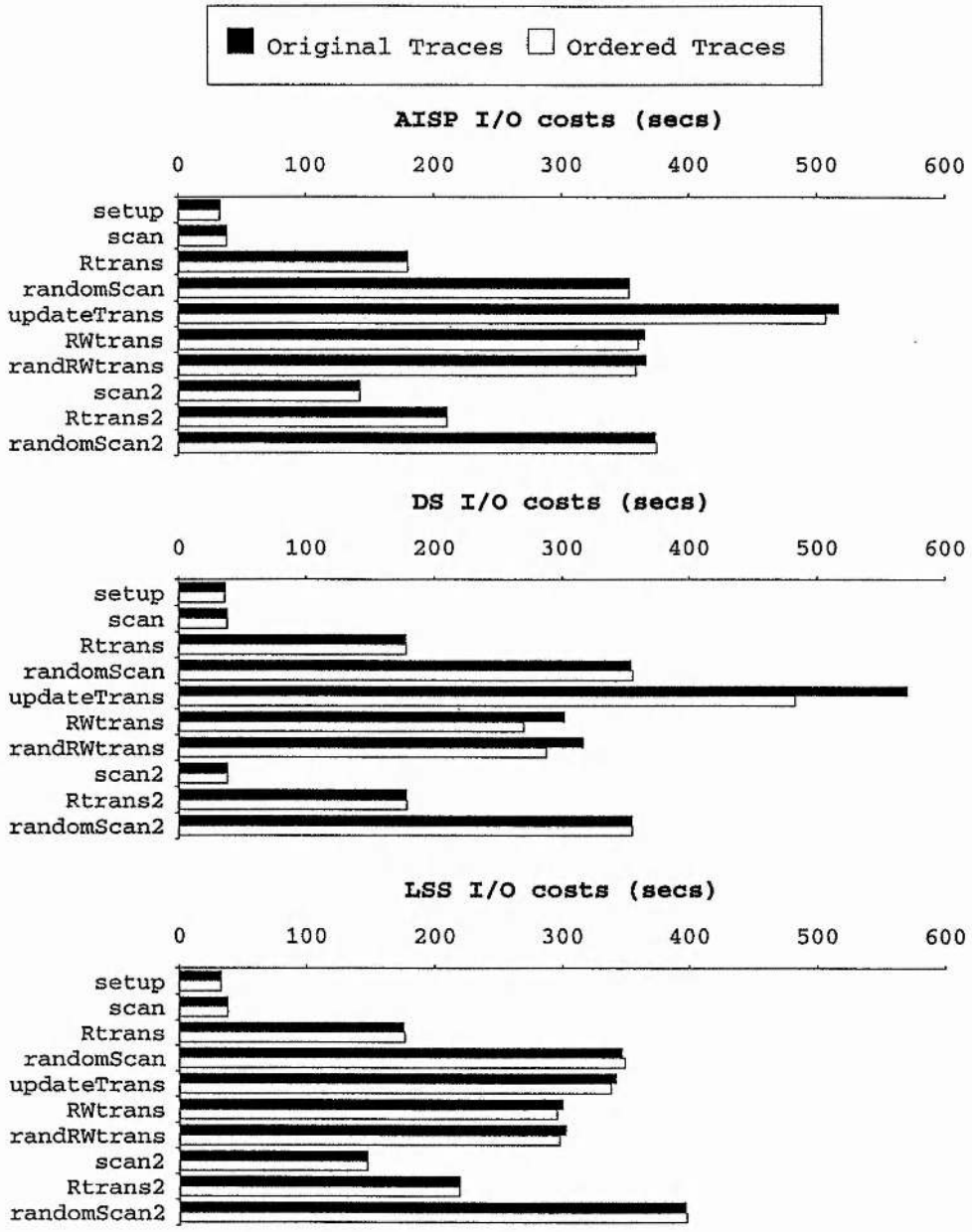


Figure 6.4: Costs of the Original and Ordered I/O Traces of MOB on the Sun

Further analysis reveals that in 164 of the 192 cases, the cost of running the ordered I/O traces generated from any two recovery mechanisms for a given workload reflect the relative costs of running the corresponding original I/O traces. For example, the costs of running the original I/O trace generated from AISP executing `readTrans2` (MOB) on the configuration of the Alpha (135.01 seconds) is lower than the cost of the original I/O trace generated from the LSD (155.88 seconds). This is also true of the costs of running the corresponding ordered I/O traces (134.72 and 155.71 respectively).

Out of the 28 results where the costs of the two ordered I/O traces do not reflect the relative costs of the two original I/O traces, 23 of the cases may be ignored since there are less than 2% variation in the costs of each pair of original I/O traces. In other words, the difference between the costs of the original I/O traces is sufficiently small that it does not matter that the ordered traces do not reflect order of the original traces.

The 5 remaining results may be accounted for by similar reasons to why the ordered I/O trace and the original I/O trace of `updateTrans` (MOB) on the Sun vary significantly.

The results verify that for most of workloads executed, there is no significant variation between the cost of running an original I/O trace and the corresponding original I/O trace. The justification of this hypothesis strengthens the approach used in MaStA to calculated I/O costs, i.e. each I/O operation performed by a recovery mechanism is categorised and the cost of each category is calculated independently.

6.5 Validation of the Access Pattern Cost Assumption

The requirement of this procedure is the justification of the hypothesis (the Access Pattern Cost Assumption of MaStA):

> To make predictions of the relative costs of recovery mechanisms for all workloads, it is sufficient to assign a predicted average cost to each I/O access pattern.

Each operation in the original I/O traces recorded in Section 6.4 is assigned the appropriate predicted I/O cost according to the predicted I/O access pattern performed (Figure 6.5). For example, in DataSafe, log writes are believed to be performed sequentially and so each log write recorded in a trace generated from DataSafe is assigned the predicted cost of a *sequential* write. The predicted costs of the I/O access patterns on the configurations of the Sun and the Alpha are measured as described in Section 4.4.1 (Appendix B). Assigning a predicted cost to each operation recorded in the I/O traces results in a predicted I/O cost for each workload running on each recovery mechanism and platform.

The hypothesis is justified if for each pair of recovery mechanisms, where the variation in the total costs of executing a given workload is significant, the predicted I/O costs can be used to select the mechanism that incurs the lower total cost. Similarly to Section 6.3, the variation in the total costs of two mechanisms is considered significant if the variation is greater than 5% of the lower total cost. There are 103 such variations in the workloads executed.

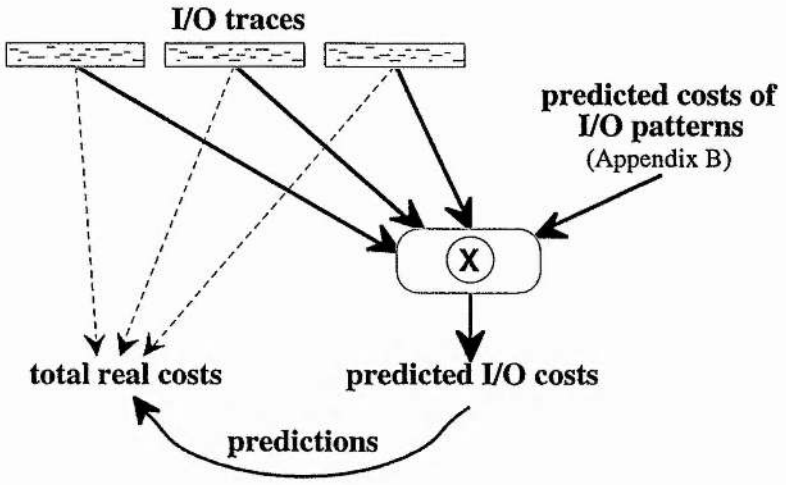


Figure 6.5: The Strategy Used to Validate the Access Pattern Cost Assumption

6.5.1 Results

The predicted I/O costs and the total real costs used in this validation procedure are given in Appendix C.3. Analysis indicates that in 100 of the 103 comparisons of recovery mechanisms where there is a significant total cost variation, the predicted I/O costs could be used to predict which mechanism incurs the lower total cost.

The three inaccurate total cost predictions result from comparing AISP and DS executing both `insert` (OO1b) and `Q8` (OO7) on the configuration of the Sun, and from comparing DataSafe and the LSD executing `T6` (OO7) on the configuration of the Alpha. These results may be accounted for by the fact that these workloads access data that has not been updated. For example, in OO1b on the configuration of the Alpha the prediction that DataSafe incurs lower total costs than AISP for the `insert` query is incorrect (Table 6.1). This is because in MaStA, database reads incurred by AISP are assigned *unclustered* costs under the assumption that the original clustering of pages is lost, when in fact AISP also incur *clustered* database reads in this particular workload. *Clustered* database reads are incurred because `insert` is the first update query executed against the OO1b database and hence the original clustering of the data accessed by `insert` has not yet been lost. This causes the predicted I/O costs of AISP to be higher than the predicted cost of DataSafe. If AISP is assigned both *clustered* and *unclustered* database reads for `insert` the predicted cost of the mechanism is lower than the predicted cost of DataSafe and a correct prediction is made. Similar reasons account for the other two inaccurate cost predictions.

If the results of this procedure are analysed for only those pairs of recovery mechanisms where there is > 13% variation in total costs, then the predicted I/O costs can be used to produce 100% accurate comparisons of total real costs.

	Sun					
	AISP		DataSafe		LSD	
Workloads	**Total Real**	**Pred. I/O**	**Total Real**	**Pred. I/O**	**Total Real**	**Pred. I/O**
lookup (OO1b)	1309.86	1365.05	1323.71	1191.78	1316.94	1448.49
scan (OO1b)	25.14	25.68	24.93	22.45	25.08	27.23
traverse (OO1b)	80.20	84.33	80.93	73.64	80.30	89.46
insert (OO1b)	**85.89**	**82.04**	**95.07**	**81.87**	**76.63**	**74.13**
insertLarge (OO1b)	784.19	734.25	870.97	764.76	698.33	644.45
update (OO1b)	688.81	646.44	717.13	698.26	590.35	556.92
lookup2 (OO1b)	3159.59	3089.97	2893.52	2697.57	3696.46	3278.86
scan2 (OO1b)	73.64	63.33	52.45	55.29	81.64	67.19
traverse2 (OO1b)	89.81	83.02	79.70	72.50	104.66	88.07

Table 6.1: Predicted I/O Costs (secs) and Total Real Costs of OO1b on the Sun

The results verify that to make qualitatively accurate cost comparison using MaStA, of recovery mechanisms executing workloads that exhibit significant total cost variations, it is sufficient to assign an average cost to each I/O access pattern.

6.6 Validation of the Workload Assumption

The requirement of this procedure is the justification of the hypothesis (the Workload Assumption of MaStA):

> The cost of running the I/O stream generated by an application is approximately the same as running the I/O stream generated by the workload abstraction.

This procedure essentially validates that workload is correctly modelled. The strategy used to validate this hypothesis is illustrated in Figure 6.6. The workloads generated from OO1, OO1b, OO7 and MOB are characterised by a number of workload variables. These variables are used to drive a synthetic workload generator that produces workloads with similar numbers of data reads and writes, and similar locality properties to the original applications. The I/O costs of executing the synthetic workloads (synthetic I/O costs) on each recovery mechanism are measured. These costs are compared with the total real costs of the original workloads recorded in Section 6.3.

The hypothesis is justified if for each pair of recovery mechanisms, where the variation in the total costs of executing a given workload is significant (> 5%), the synthetic I/O costs can be used to select the mechanism that incurs the lower total cost. There are 103 such variations in the workloads executed.

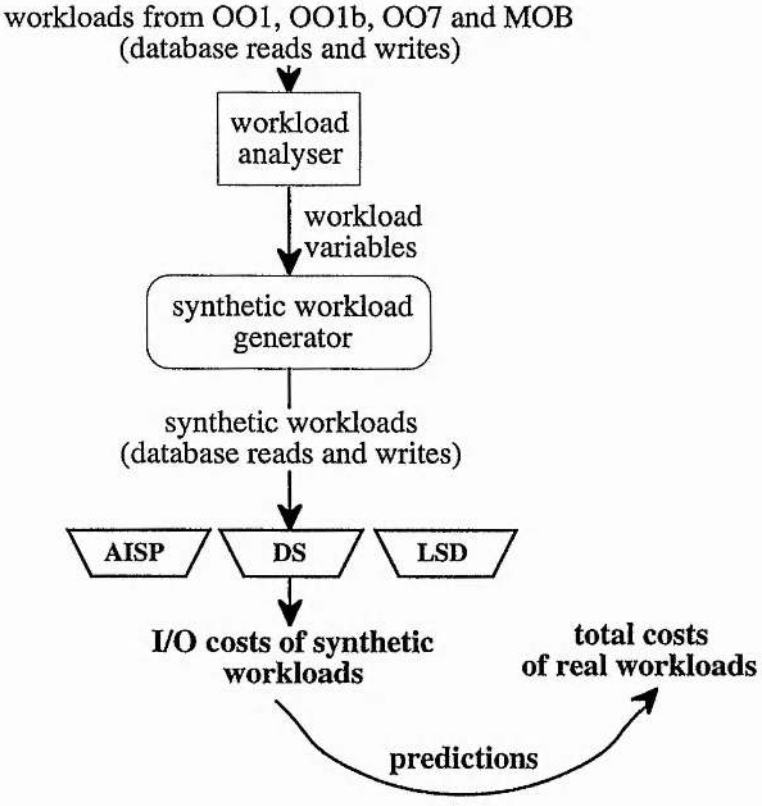


Figure 6.6: The Strategy Used to Validate the Workload Assumption

6.6.1 Characterising Workload

The number of variables used to characterise workloads are kept to a minimum to ensure that the design and implementation of the synthetic workload generator are tractable. At the same time the variables have sufficient expressive power to ensure that the synthetic I/O costs are accurate enough to predict the relative total costs of recovery mechanisms for a given workload. The variables used to characterise workloads are given in Table 6.2. The workload analyser makes use of the variables *cache* and the knowledge that the recovery mechanisms employ LRU page replacement strategies, to calculate the values of *read* and *readRecent*.

In the definitions of *readFaultLoc*, two logical database pages are considered near to one another if they are less than 1920 logical pages (15 MB) apart. This value is chosen to reflect the size of the disk partition used to measured *clustered* I/O (Appendix B).

Note that the variables used here assume that transactions are executed serially, as is the case in the workloads used in this validation procedure. Applications exhibiting concurrent behaviour may be accommodated by adding transaction behaviour variables to the workload abstraction. These may be, for example, the average number of concurrently executing transactions and the average number of concurrent transactions that access and update the same page.

Workload Variables	Description
read	the number of read operations performed
readRecent	the number of *reads* that access data already in the cache (no page faults incurred)
readFaultLoc	the number of page faults in which the database page accessed is logically near the previously faulted page
update	the number of write operations performed
firstUpdate	the number of read operations performed before the first write operation
updateTrans	the sum of the number of *update* performed by each transaction on pages already updated by the transaction
updateTemp	the number of pages updated by a transaction that have been updated by a previous transactions
commit	the number of commit operations
db	the size of the virtual database in bytes
cache	the size of the cache in bytes
page	page size in bytes

Table 6.2: Workload Variables Used to Characterise Workloads

6.6.2 Synthetic Workload Generator

The synthetic workload generator takes as input, values for the variables in Table 6.2 and produces workloads consisting of database access, update and commit operations. The generator uses a probabilistic approach to determine whether each access generated is a read or write, and to select the database page accessed by each operation.

- An operation is a read if the number of operations generated so far in a workload is $< firstUpdate$. Otherwise, an operation has a $read/(read + write)$ probability of being a read, otherwise it is a write.

- If a read operation is generated, the probability that the page accessed by the operation has been read recently is $readRecent/read$, and hence the operation does not cause a page fault. If a read operation is generated to cause a page

fault, the probability that the faulted page is near the previously faulted page is *readFaultLoc*/(*read* - *readRecent*).

- If a write operation is generated, the probability that the operation changes a page already updated by the current transaction is *updateTrans*/*update*. If so, a page already updated by the transaction is randomly selected. If not, the probability that the operation updates a page changed by a previous transaction is *updateTemp*/(*update* - *updateTrans*).

- A commit operation is performed every ((*read* + *update*)/*commit*) operations.

The standard library function `random` was used to produce the random values required by the synthetic workload generator.

6.6.3 Results

The average I/O costs of the synthetic database workloads and the total costs of the original workloads executing on the three recovery mechanisms and the two platforms are given in Appendix C.4. The costs of each pair of mechanisms executing a given workload are analysed to determine if the relative order of the synthetically produced I/O costs reflect the relative order of the total real costs. Analysis indicates that in 101 of the 103 comparisons of recovery mechanisms, the synthetic I/O costs could be used to predict which mechanism incurs the lower total cost.

The two inaccurate predictions occur when AISP and DataSafe executing `update` (OO1b) on the configuration of the Alpha are compared and when the same mechanisms executing `lookup2` (OO1b) on the Sun are compared. No satisfactory explanation can be found for these two results. In future work, such results may be corrected by incorporating more workload variables, for example, to develop a more accurate model of workload locality.

If the results of this procedure are analysed for only those pairs of recovery mechanisms where there is > 10% variation in total costs, then the synthetic I/O costs can be used to produce 100% accurate comparisons of total real costs.

6.7 Accuracy of MaStA

Having validated the assumptions of MaStA, a final procedure is performed to show that the costs produced using the model are sufficiently accurate to provide good qualitative comparisons of the costs of recovery mechanisms. In other words this procedure is required to verify that mechanism, application workload and platform are accurately modelled in MaStA. The strategy used in this procedure is illustrated in

Figure 6.7. The workload variable values measured in Section 6.6, and the average cost of each I/O pattern recorded in Appendix B are used to drive the MaStA cost models of AISP, DataSafe and the LSD developed in Chapter 3. The resulting I/O cost predictions are analysed to determine if for each workload and for each pair of recovery mechanisms where there is > 5% variation in total costs of executing the workload, the predicted I/O costs can be used to select the mechanism with the lower total cost.

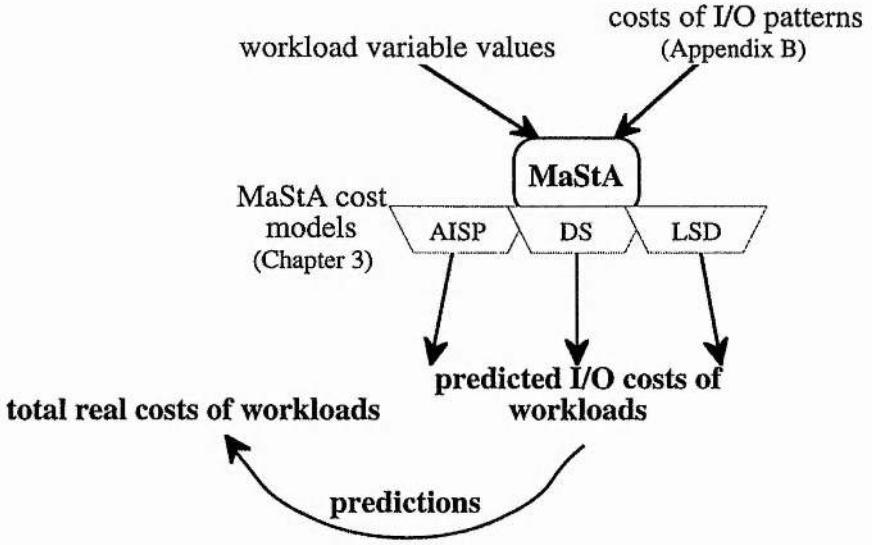


Figure 6.7: The Strategy Used to Show the Accuracy of MaStA

6.7.1 Results

Appendix C.5 gives the average real total costs and the I/O cost predictions made using MaStA configured for the Alpha, the Sun and with a uniform I/O cost. Analysis indicates that in 102 of the 103 comparisons of recovery mechanisms where there is a significant total cost variation, the predicted I/O costs could be used to predict which mechanism incurs the lower total cost.

The failure occurs when the costs of DataSafe and the LSD executing T6 (OO7) on the configuration of the Alpha are compared - the prediction that DataSafe incurs lower total costs than LSD is incorrect. In MaStA, database reads incurred by the LSD are assigned *disk* costs, when in fact the LSD also incurs *clustered* database reads in this workload since T6 accessed data which has not yet been updated. This causes the predicted I/O cost of the LSD to be higher than the predicted cost of DataSafe. If the LSD is assigned *clustered* database reads for T6 the predicted cost of the mechanism is lower than the predicted cost of DataSafe and a correct prediction is made.

Further examination of the results highlights the necessity to configure I/O access patterns against the platform being used by the fact that the same database workload

may suit different recovery mechanisms on different platforms. For example, the best recovery mechanism on the configuration of the Sun for the `RWtrans` (MOB) query is the LSD, whereas the best mechanism for this workload on the configuration of the Alpha is DataSafe.

The cost comparisons made using the predicted I/O costs are 100% accurate for pairs of recovery mechanism where the total cost variation is greater than 6% of the lower total cost.

6.7.2 Comparison with Uniform Cost Models

Early analytical models of recovery mechanisms use uniform I/O costs to predict the costs of recovery mechanisms. The MaStA model on the other hand is designed to make cost predictions taking into account the differences between the costs of various I/O access patterns. The accuracy of this technique is highlighted by comparing the results of this procedure with MaStA cost predictions made using a uniform I/O cost (Appendix C.5). When each access pattern in MaStA is assigned a uniform cost the accuracy of the resulting predictions are poor. In fact, for each pair of recovery mechanisms where there is a significant total cost variation of executing a given workload, the mechanism with the lower total cost is predicted in only 35 of the 103 comparisons. The poor results are caused by AISP and the LSD performing the same number of I/O operations for all workloads. Assigning a uniform cost to these operations results in equal cost predictions for the mechanisms, thus providing no useful comparisons. Furthermore, for all update workloads used in this procedure DataSafe performs higher numbers of I/O operations than AISP and the LSD. Therefore in a uniform I/O cost model DataSafe is always predicted to incur the highest I/O costs.

6.7.3 Conclusions

The requirement of this procedure is to show that the costs produced using the MaStA I/O cost model can be used to provide good qualitative predictions of the I/O costs of recovery mechanisms. The results indicate that this is the case for the majority of workloads where total cost variations on different recovery mechanisms are significant.

6.8 Conclusions

In this chapter the assumptions that support the abstractions of MaStA are justified by four validation procedures. The procedures execute database workloads generated from a number of benchmarks and synthetically generated workloads on various

recovery mechanisms and various platform configurations. The CPU and I/O costs of the workloads are measured and traces of the database accesses and I/O operations performed are recorded. Justification of the assumptions consists of analysing these costs and traces for each assumption. The results of the analysis suggest that each assumption holds for the majority of workloads where there are significant variations in the total costs of using different mechanisms.

A distinguishing feature of the MaStA model is that it differentiates between various patterns of I/O accesses. The necessity to distinguish between I/O access patterns is highlighted by comparing predicted costs produced using MaStA configured for real platforms against costs predicted using a uniform I/O cost. When the model is configured with a uniform cost it cannot distinguish between the costs of mechanisms that perform the same number of I/O operations. The importance of distinguishing I/O access patterns is further highlighted by the fact that the best mechanism for a particular workload may vary across different platforms, depending on the relative costs of the I/O access patterns.

By justifying the assumptions and illustrating that MaStA can produce sufficiently accurate cost comparisons of recovery mechanisms, this chapter has shown that MaStA is suitable for use in a flexible database architecture such as Flask to choose the mechanism that incurs the lowest cost for a given application and platform.

7 Worked Example of the Flexible Architecture

7.1 Introduction

In Chapter 4 a new analytical model for recovery mechanisms called MaStA was described. A worked example is now provided to illustrate how MaStA may be used to choose an appropriate recovery mechanism. The example describes the design of a database of information and two applications that use the information. MaStA is used to configure two instantiations of Flask on which the applications are executed. This involves characterising the database and each application using MaStA's workload variables, configuring the model against the execution platforms and selecting the mechanism with the lowest predicted I/O costs. The applications are also executed on each mechanism and measured to verify the choices of mechanisms.

7.2 Scenario

A bank and a building society each wish to maintain a database of information about customers indexed by account number. For each customer, the database must record a name, a date of birth, an address, an account balance and for security purposes an image. Each database will be maintained by a server to which multiple clients may send transactions to be executed serially. The databases are required to record information on 65000 customers. Figure 7.1 depicts the scenario.

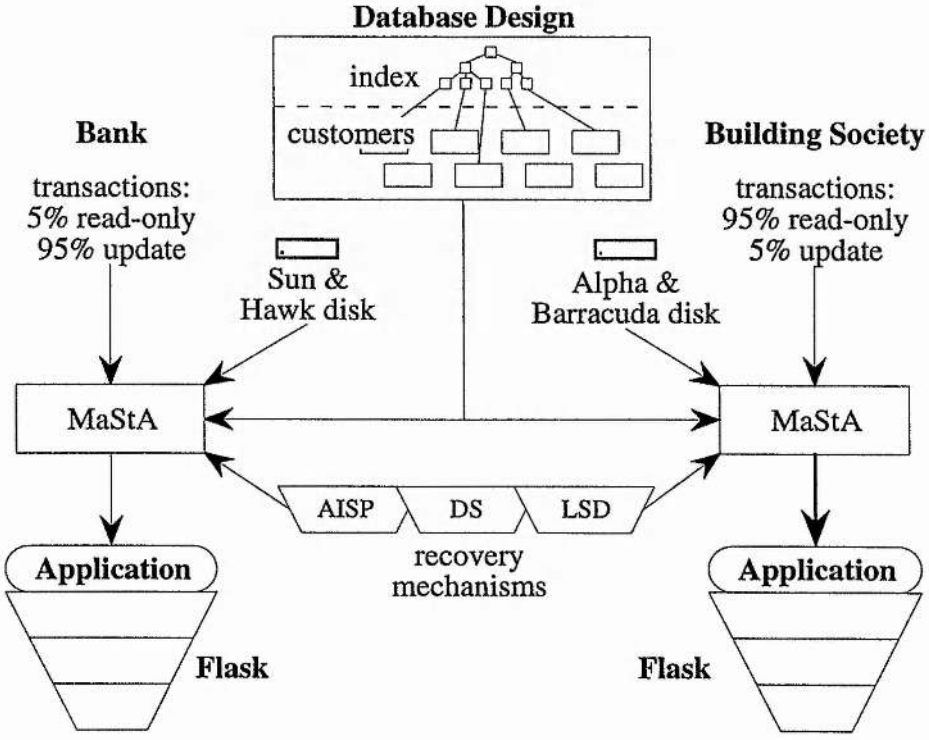


Figure 7.1: Using MaStA in a Worked Example

Each company has provided a prediction of the style of transactions that will execute over its database. The building society predicts that 95% of transactions executing against its database will be read-only and each transaction will retrieve information about a single customer chosen at random. The remaining 5% will update the balances of two customers chosen at random. The bank predicts that 5% of transactions will be read-only transactions and 95% will be update transactions. The two applications are designed in this scenario to exhibit widely varying workloads to emphasise the effectiveness of MaStA to choose the appropriate recovery mechanism for different workloads.

The database servers are implemented in Napier88 and executed on the Flask architecture to take advantage of the flexible recovery management. Flask is configured with AISP, DataSafe or the LSD mechanism developed in Chapter 3. MaStA is used to choose between these mechanisms for each application.

The bank has a Sun SPARCStation ELC running SunOS 4.1.3 with 48 MB main memory, a 500 MB CDC Wren V SCSI drive dedicated to the operating system and a 500 MB partition on a 2.1 GB Seagate ST32151N Fast SCSI-2 (Hawk 2XL). The building society has a DEC Alpha AXP 3000/600 running OSF/1 V3.2 with 128 MB main memory, a 1 GB DIGITAL RZ26 SCSI drive dedicated to the operating system and a 500 MB partition on a 2.1 GB Seagate ST12550N SCSI drive (Barracuda II). These particular platforms are chosen for this scenario since the MaStA I/O access patterns (sequential read, ordered write, etc.) have already been measured (Appendix B). If other platforms had been used, MaStA would have been calibrated against them by performing the I/O access pattern experiments described in Section 4.4.1.

7.3 Database Design

The databases accessed by the bank and the building society are composed of a number of customer records indexed using a B+tree. Each customer is represented by a Napier88 structure instance of type:

```
type Customer is structure
(   balance : int ;
    name    : string ;
    address : string ;
    picture : image ;
    age     : int )
```

Napier88 creates five objects to compose a structure instance of type `Customer`:

- customer structure instance (this includes the `balance` and `age` fields)
- name string
- address string
- image descriptor
- image bitmap

Knowledge that each `Customer` is represented by five objects is used in Section 7.4 to characterise the workloads of the company's applications. Each node of the B+tree used to index customers is created from an instance of the type:

```
rec type Node is structure
(   entries : int ;             !number of subtrees
    leaf    : bool ;            !indicates a leaf node
    indices : *int ;            !a vector of index values
    pointers: *Pointer)         !a vector of pointers to
                                !variants of type Pointer
&
Pointer is variant
(   next    : Node ;            !points to either another
    value   : Customer)         !B+tree Node or to a Customer
```

Three objects are created in Napier88 to compose each node of the B+tree: a `Node` structure instance, a vector of integers index values and a vector of `Pointer` variants to point to either `Node` or `Customer` values.

The database is generated by first building a B+tree sufficiently large to index 65000 customers and then creating each customer in index order. The index is built first to ensure that indexing information exhibits good spatial locality in the database thereby potentially reducing read costs for the index. An order-4 B+tree is used. This means that each node of the B+tree contains between 3 and 7 pointers to other nodes or to customer records. Using an order-4 B+tree requires approximately 16000 nodes to index the 65000 customer records.

The layout of the database is illustrated in Figure 7.2. The sizes of the areas of the database required to hold the various objects are estimated from the numbers and sizes of objects created.

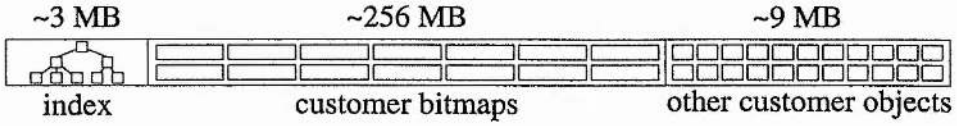


Figure 7.2: Layout of Objects in the Database

7.4 Characterising Workloads

To produce I/O cost predictions using MaStA for AISP, DataSafe and the LSD, the applications are analysed to determine the values assigned to the MaStA workload variables described in Table 4.3.

7.4.1 The Building Society's Workload

The predicted workload of the building society is analysed using 40000 transactions. This number of transactions is assumed to be sufficiently large to accurately represent the characteristics of the application when it is executing continuously on the three recovery mechanisms available. The analysis is broken down by calculating the contribution to each workload variable made by three sets of objects: indexing information, customer bitmap objects and the remaining objects composing customer records. Analysis is performed in this manner to reflect the layout of these objects in the database (Figure 7.2).

Since the B+tree index is accessed frequently, it is assumed that each page (8 Kbytes) containing nodes of the index is faulted only once and remains in the database cache (8 MB). Hence it is assumed that the minimum number page faults, i.e. 384 faults, are incurred when accessing the 3 MB (384 pages) of indexing information, and that good locality (90%) is observed of these faults. The workload values attributed to reading indexing information are estimated to be:

read = 48000 (16000 Nodes * 3 objects)
readRecent = 47616 (*read* - page faults)
readFaultLoc = 346 (90% of page faults)

A total of 42000 customers are accessed (95% of 40000 transactions * 1 customer + 5% of 40000 transactions * 2 customers), requiring 42000 *read* to access the corresponding bitmap objects. Since accesses to the bitmaps are sparse, *readRecent* is assigned 0, i.e. each bitmap access causes a page fault.

The `customer` structure instance, name, address and image descriptor objects of a customer contribute 168000 to *read* (42000 customers * 4 objects). The objects of a particular customer are assumed to reside on the same page. Therefore *readRecent* is

assigned 126000 ($^3/_4$ * *read*) since three out of four *read* for each customer, access the same page. Since a number of customer records reside on each database page some degree of temporal locality (10%) is assumed in customer accesses. Therefore an additional 4200 (10% of 42000 customers) is assigned to *readRecent*.

Due to the unclustered nature of this workload, it is assumed that the degree of spatial locality of customers accessed is poor: only 10% of page faults are clustered. Hence customer bitmap accesses contribute 4000 to *readFaultLoc* (10% of *read* - *readRecent* for bitmaps). The value of *readFaultLoc* for accessing other customer objects is 3360 (10% of *read* - *readRecent* for the other customer objects).

The building society predicts that 5% of transactions update the balances of the customer records they access. It is assumed that the objects containing the balances of the two customers updated by each transaction are held on different pages and hence two pages are updated by each transaction. An additional three pages are updated for each transaction commit to record Napier88 overheads.

update = 10000 (2000 update transactions * 5 pages)
updateTrans = 0

Workload Variables	Values
read	258000 pages
readRecent	177816 pages
readFaultLoc	8018 pages
update	10000 pages
updateTrans	0 pages
updateLoc	5%
commit	2000
propWrite	4488 pages
propWriteFinal	512 pages
page	8192 bytes
mapEntry	8 bytes
root	1
db	270 MB

Table 7.1: Variable Values for the Building Society's Workload

The temporal locality of the pages updated by transactions is assumed to be high since these pages contain customer balances that are frequently updated, and hence infrequently chosen for replacement in the cache. Therefore it is assumed that a high proportion of pages (50%) updated in the cache by transactions are updated again by

other transactions before being propagated to the database in the DataSafe mechanism.

propWriteFinal = 512
propWrite = 4488 ((50% of update) - *propWriteFinal*)

The value of *propWriteFinal* is estimated to be 50% of the size of the database cache (1024 pages) used by the recovery mechanisms. Table 7.1 provides a summary of the workload variable values used to characterise the predicted database workload of the building society.

7.4.2 The Bank's Workload

The database workload predicted by the bank is analysed using 20000 transactions, as opposed to the 40000 transactions of the building society to make the workloads generated from the two applications comparable. The workload variable values for reading indexing information in the bank's application are assumed to be similar to those of the building society's.

A total of 39000 customers (5% of 20000 transactions * 1 customer + 95% of 20000 transactions * 2 customers) are accessed with similar degrees of temporal and spatial locality to the building society's predicted workload. The workload values attributed to reading indexing and customer information are:

read = 48000 (for index objects)
+ 39000 (for customer bitmaps)
+ 156000 (39000 customers * 4 objects)

readRecent = 47616 (for index objects)
+ 0 (for customer bitmaps)
+ 120900 (39000 customers * 4 objects * $^3/_4$ +
10% of 39000 customers)

readFaultLoc = 346 (for index objects)
+ 7253 (10% of *read-readRecent* for all customer objects)

In the bank's predicted workload 95% of transactions update the balances of the customer records they access and similarly to the building society's workload each transaction updates 5 pages.

update = 95000 (19000 update transactions * 5 pages)
updateTrans = 0

Workload Variables	Values
read	243000 pages
readRecent	168516 pages
readFaultLoc	7448 pages
update	95000 pages
updateTrans	0 pages
updateLoc	5%
commit	19000
propWrite	46732 pages
propWriteFinal	768 pages
page	8192 bytes
mapEntry	8 bytes
root	1
db	270 MB

Table 7.2: Workload Variable Values for the Bank's Workload

Similarly to the building society's workload 50% of pages updated in the cache by transactions are assumed to be updated by other transactions before being propagated to the database in the DataSafe mechanism.

propWriteFinal = 768
propWrite = 46732 ((50% of *update*) - *propWriteFinal*)

The value of *propWriteFinal* is estimated to be 75% of the size of the database cache (1024 pages) - a higher percentage than the building society since a higher proportion of the pages accessed are updated. The values assigned to the workload variables for the bank are summarised in Table 7.2.

7.5 Utilising MaStA

In addition to providing values for the MaStA workload variables, the I/O access patterns used in MaStA must be configured against the platforms on which the databases are maintained. In this scenario the companies make use of the configurations of the Sun and the Alpha employed in earlier chapters. Therefore the I/O access patterns of MaStA are configured with the values recorded in Appendix B.

Table 7.3 and Figure 7.3 give the I/O costs obtained from evaluating MaStA for AISP, DataSafe and the LSD using the workload variable values given in Tables 7.1 and 7.2 and the I/O access pattern costs of Appendix B. The workload functions used are those developed in Chapter 4. The results suggest that for the bank, Flask should be configured with the LSD to provide the best performance and that for a marginal

gain in performance the architecture should be configured with DataSafe for the building society. Table 7.3 also gives the I/O costs obtained for the two applications when MaStA is configured with a uniform I/O cost. The values in Table 7.3 highlight that the use of a uniform I/O cost generates cost predictions that do not distinguish between mechanisms that incur the same number of I/O operations (AISP and the LSD).

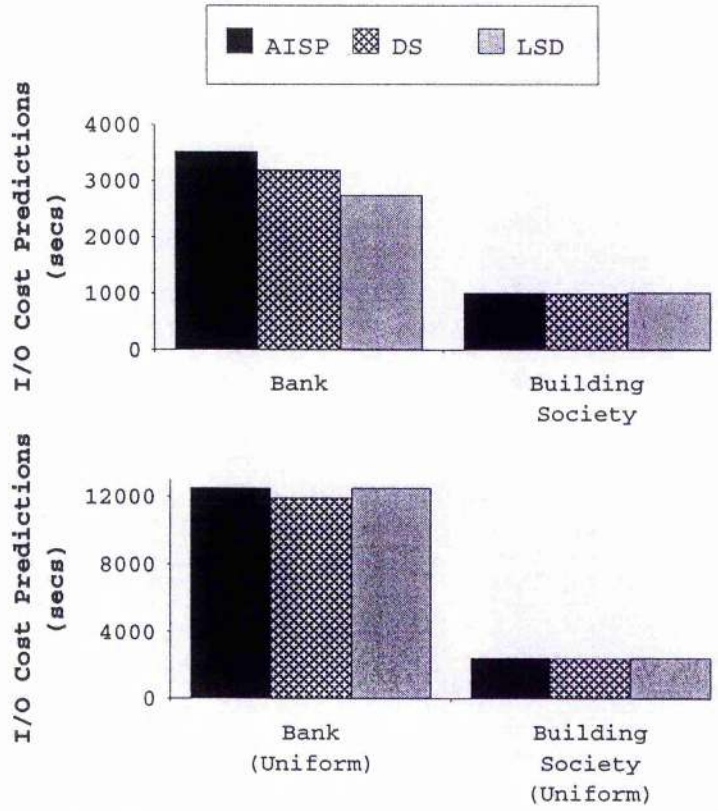


Figure 7.3: Predicted I/O Costs (seconds) Calculated Using MaStA

Application	AISP	DataSafe	LSD
Bank	3534	3197	2752
Building Society	1020	1007	1027
Bank (Uniform)	7153	6825	7153
Building Society (Uniform)	2443	2410	2443

Table 7.3: Predicted I/O Costs (seconds) Calculated Using MaStA

7.6 Verification of Cost Predictions

To verify the choices of mechanisms made using MaStA, the workloads were generated and executed on each recovery mechanism available. The database and the applications used to generate the workloads are implemented in Napier88 and executed on instantiations of Flask configured with the different recovery mechanisms

available. The code for maintaining the B+tree index, building the database and the code for the two applications that generate the workloads are included in Appendix D. The bank's application executes 20000 transactions and the building society's application executes 40000 transactions, the same numbers of transactions as analysed in Section 7.4.

For each recovery mechanism, the database is built and each application executed six times. The elapsed execution time of each application is averaged over the last three executions of the application. Only the last three executions are taken into account to ensure that any effects on I/O costs of the recovery mechanisms are shown in the results. Elapsed execution costs are measured using the UNIX/OSF `time` command and with the platforms in single user mode. Table 7.4 contains the average total execution costs in seconds of each application executing on the three recovery mechanisms.

Application	AISP	DataSafe	LSD
Bank	5314	4249	3736
Building Society	1597	1499	1575

Table 7.4: Total Real Costs (seconds) of the Applications

The results concur that the LSD should be used to provide the best performance for the bank's application and that DataSafe should be used for the building society. Furthermore, the results confirm that the next best mechanism for the bank is DataSafe. On the other hand the predicted costs (Table 7.3) are not sufficiently accurate to predict that the next best mechanism for the building society is the LSD which incurs lower costs than the AISP mechanism. Since there is only a marginal variation in the total costs of using these two mechanisms for this particular workload (1.4%) there would be no significant effect on the performance of the building society's database of choosing the LSD over the AISP mechanism.

7.7 Conclusions

Previous chapters have introduced and validated a new analytical cost model for recovery mechanisms called MaStA. This chapter has attempted to illustrate the utility of the model in the flexible Flask architecture and to promote confidence that MaStA produces sufficiently accurate cost predictions to be effective in such an architecture.

A scenario is described in which two companies predict the transaction workloads that will be executed on their databases held on different platforms. Analysis of the workloads is performed by studying the layout of the database and estimating the values that should be assigned to the MaStA workload variables. MaStA is then used

to predict with which recovery mechanisms Flask should be configured to provide the best performance for each application. The total cost of executing each workload on each mechanism available is then measured. Analysis of the real and predicted costs indicates that MaStA predicts the I/O costs of the schemes with sufficient accuracy to choose the mechanism that incurs the lowest total cost for each application.

The use of MaStA to successfully configure recovery management to obtain good performance goes some way towards validating the thesis that analytical techniques may be employed to configure database management systems to increase performance.

8 Conclusions

The rapid expansion of electronic commerce and communications have put ever increasing demands for performance on computer systems. How can one determine if a system is executing efficiently? The many layers of abstraction present in modern systems - the application, operating system, networks, platform - make understanding the behaviour of such systems a complex task. Past studies have used empirical measurement techniques [KGC85, CS92, CDN93] on executing systems to determine whether optimisations enhance performance. Another technique is to develop simulations of systems to predict behaviour. Both empirical and simulation based analysis tend to be expensive in terms of programming, debugging and validation. A cheaper and less time consuming alternative is to employ analytical modelling to predict performance [Reu84, AD85].

The thesis of this work is that analytical modelling can be used to accurately configure recovery in database management systems to provide optimum performance for any application and platform. A new analytical model was developed to compare recovery mechanisms. In addition, two new recovery mechanism were incorporated into an existing flexible architecture to provide a basis on which the model could be validated. Validation of the model involved executing synthetic database workloads over various mechanisms and, by analysis of the results obtained, justifying the assumptions which underly the model. The utility of the model was then illustrated by a worked example in which MaStA was used to configure Flask to provide the best performance for different database applications.

8.1 Cost Prediction

In order to configure recovery management in a DBMS a technique is required that allows the costs of recovery mechanisms to be compared for any application and platform. This work adopted an analytical approach to provide cost predictions of recovery mechanisms. The MaStA I/O cost model presented increases the accuracy of cost predictions over existing models by taking into account variations in the patterns of I/O accesses performed by recovery mechanisms such as the difference between sequential and synchronous unclustered I/O. This is in contrast to early studies of recovery mechanisms which often used a uniform I/O cost. MaStA divides the problem of producing cost predictions into three abstractions:

- the behaviour of recovery mechanisms;
- workload characteristics;

- and platform characteristics.

The behaviour of each recovery mechanism is captured in the model by categorising the I/O operations incurred in terms of the movement of data between a database, a cache and a log. By assigning the appropriate I/O access patterns to each category dependent on the characteristics of the mechanism, the model ensures that the cost of each category is configured for the mechanism. The number of I/O operations incurred in each category is estimated using a workload model that takes into account application characteristics that affect I/O. The accuracy of the model is attained by calibrating the cost of each I/O pattern against the platform on which the application is executed thereby ensuring that cost predictions are platform specific.

The modelling techniques used in MaStA are dependent on four assumptions. These are:

- In applications where variations in total costs of using different recovery mechanisms are significant, the variations in the CPU costs incurred are insignificant compared to the variations in the I/O costs.
- The interaction between the different categories of I/O accesses is not significant; that is, the cost of running the I/O stream generated by a given recovery mechanism is not significantly different from the sum of the costs of running the streams of each I/O cost category separately.
- To make predictions of the relative costs of recovery mechanisms for all workloads, it is sufficient to assign a predicted average cost to each I/O access pattern.
- The cost of running the I/O stream generated by an application is approximately the same as running the I/O stream generated by the workload abstraction.

These assumptions were validated on a flexible architecture to ensure the accuracy of the modelling techniques used.

8.2 Flexible Architecture

The Flask architecture was extended so that recovery management could be configured with any one of a number of different mechanisms for a given application. This provided a basis on which MaStA validation experiments could be performed and provided an opportunity to illustrate the utility of MaStA in a flexible architecture.

Flask is an architecture that provides opportunities to independently configure concurrency and recovery. This is achieved by separating these components in a layered design in which concurrency is modelled in terms of the movement of data between access sets and the database. Recovery management assumes that concurrency control is performed at a higher layer in Flask and is responsible for providing the access sets using any implementation.

The two new mechanisms developed for Flask in this work are DataSafe and a log-structured mechanism, either of which can be used as an alternative to the after-image shadow paging mechanism used in the first instantiation of Flask. DataSafe is a page-based logging mechanism that exhibits considerable differences in behaviour from those exhibited by the after-image shadow paging mechanism, and like shadow paging is independent of concurrency ensuring that different models of concurrency may be provided in Flask. The design of the log-structured mechanism differs from the AISP in that writes to the log are performed in a sequential manner as opposed to the shadow paging mechanism which performs writes in an ordered fashion to non-contiguous blocks.

8.3 Validation

The validation strategy employed in this work was designed to verify that the assumptions of MaStA are valid for a number of applications and recovery mechanisms executing on various platform configurations. The strategy employed Flask to execute database workloads generated from Napier88 using different recovery mechanisms thus providing the opportunity to accurately compare the mechanisms under identical workloads. The costs incurred by the mechanisms were measured and traces of the I/O operations performed were recorded. These results and traces were then analysed to validate the assumptions. The results of these analyses suggested that the assumptions hold for the workloads, recovery mechanisms and platforms employed. The validation analyses also highlighted the requirement to distinguish between different patterns of I/O, by comparing results obtained from MaStA configured for real platforms against the model configured with a uniform I/O cost. It was found that in the latter case no accurate distinction could be made between the recovery mechanisms employed in the validation strategy.

Having validated the assumptions of MaStA and promoted confidence that it produces sufficiently accurate cost predictions the utility of the model within Flask was illustrated by proof of concept. The scenario involved configuring Flask's recovery manager to provide the best performance for two database applications. The example discussed how the applications were analysed to characterise their workloads. MaStA

was then configured against the platforms employed to execute the applications and was used to predict the costs of each mechanism from which a choice of mechanism was made for each application. The accuracy of the choices were verified by configuring Flask with each mechanism available and measuring the execution of the applications on each configuration. The results confirmed that MaStA is sufficiently accurate to configure recovery management to provide optimum performance for a given application and platform.

An observation that has come from validating the MaStA model is that the process of performing experiments on real systems is both time and resource consuming. The straightforward design of the validation strategy expanded into a test suite consisting of more than 2000 experiments. These experiments required approximately 12 months to design, program and to execute and were complicated by sources of interference each of which required numerous experiment design iterations to eliminate. Having performed the experiments an additional three months were required to analyse and interpreter the results, and to determine how they should be portrayed understandably. In terms of resources, the validation strategy required two the single user platforms to execute the experiments and approximately 4 GB of disk space to hold the Napier88 system, instantiations of Flask, benchmark databases and queries, and traces.

8.4 Future Work

To reduce complexity in the initial design of MaStA the effects of concurrency were omitted. This factor will be included in future developments in the model so that applications exhibiting concurrent behaviour are accommodated. Modifications required to achieve this include the development of new I/O categories and workload variables to calculate costs such as the overheads of performing transaction aborts. The cost of recovering a database after system failure was also omitted to reduce complexity. The cost of providing for recovery during normal processing and the cost of performing recovery after failure may not be easily combined into one useful value for each application and mechanism since the relative importance of these two costs depends largely on the style of application. These costs will therefore be calculated separately allowing the costs to be analysed individually when making a choice of mechanism for a particular application. Future investigation should also include incorporating the wide range of object logging schemes used in database systems into MaStA. This may involve the design and implementation of such schemes in Flask to allow the validation of MaStA for object based mechanisms.

To show that the techniques developed in MaStA are applicable in commercial environments future work will also investigate the inclusion of commercial databases

in the validation strategy to establish that the accuracy of the model is not dependent on any attributes of Napier88. This may involve augmenting operating systems or device drivers to obtain information about the resources consumed by these systems. More challenging still will be the inclusion of architectures that make use of parallel file systems [TW95] or RAID technology [PGK88] to increase I/O throughput.

This work has focused on choosing the best performing mechanism for a particular application and linking the mechanism statically into a flexible architecture. It may also be possible to use the model in a more dynamic manner. For example it may be possible to embed the model in a recovery manager to analyse the workload of the executing application. Results from the analysis may be used either by the user or automatically to dynamically select the mechanism that provides the best performance should the workload of the application change.

The analytical techniques developed here are not restricted to configuring flexible recovery. It is conceivable that the techniques may also be used in the configuration of many other aspects of computer systems where policy decisions must be made. These may include:

- the selection of main memory and non-volatile storage garbage collection techniques based on the application's store usage;
- configuring operating system page swapping selection algorithms;
- and configuring distributed systems based on models of network message loads.

8.5 Finale

The work presented in this thesis developed an analytical model for predicting the costs of recovery mechanisms and through analysis and proof of concept demonstrated that such a technique can be used to successfully configure recovery in database management systems to provide the best performance. It is clear from this work that no one mechanism can provide the best performance for all applications but whether commercial organisations adopt such flexible approaches in their systems is still to be seen. If they do it is hoped that the techniques developed here will prove useful in configuring such systems.

Glossary

after-image. The after-image of an item is the value of the item once it has been updated.

AISP. Acronym for after-image shadow paging.

availability interval. The potential number of I/O block transfers that may be performed in the mean time between failures.

before-image. The before-image of a data item is the value of the item before it is updated.

BISP. Acronym for before-image shadow paging.

clustered I/O. These are localised accesses that are synchronous and hence cannot be ordered

DBMS. Acronym for database management system.

disk I/O. These are synchronous accesses that involve moving the access position arbitrarily far on the device.

idempotent. The property of restart that a sequence of incomplete restarts followed by a successful completion results in the same state as if the initial restart had succeeded.

LSD. Acronym for log-structured database.

materialised database. The term describes the state of a database only, i.e. taking no account of additional data which may be recorded during normal processing to recovery the database to a consistent state.

no-redo. A recovery mechanism does not require *redo* if all a transaction's updates are written to the database before or when the transaction commits.

no-undo. A recovery mechanism does not require *undo* if it does not write uncommitted updates in place in the database.

ordered I/O. These are I/O operations performed on sorted non-adjacent locations.

propagation. These are I/O operations required by some mechanisms to transfer committed data to the database.

sequential I/O. These are I/O operations performed on contiguously increasing positions.

redo. A mechanism is *redo* if it must propagate committed updates from the log to the materialised database on restart.

transaction rollback. This involves removing from a database all updates made by an aborting transaction.

unclustered I/O. These are synchronous accesses that involve moving the access position arbitrarily far within the database.

redo. A mechanism is *undo* if it must remove uncommitted updates from the materialised database on restart by copying before-images from the log.

Appendix A Recovery and Benchmark Configurations

A.1 Recovery Mechanism Configuration

Each recovery mechanism used in the validation procedures is configured with an 8 MB database cache composed of a number of cache slots. The state information of each slot is recorded in a cache map composed of two word entries.

Each mechanism employs an LRU cache page replacement algorithm. This is implemented by maintaining a flag in the cache map for each cache slot indicating whether the slot has been accessed since the previous page selection, and a count of the number of selections the slot has survived without being accessed. A cache slot's count is incremented during page selection if the page has not been accessed since the previous page selection, otherwise the count is reset to zero. During page selection the cache slot with the highest count value is chosen. Cached database pages are indexed using an external overflow hash table.

In the experiments described in Chapters 6 and 7, AISP and the LSD make use of the entire 500 MB raw partition available on each platform as a database. DataSafe splits each partition available into two: a 300 MB partition for use as the database and a 200 MB partition for use as a safe.

A.2 Benchmark Configurations

The benchmarks described in Section 5.4.2 and used in the validation procedures described in Chapter 6 have the following configurations.

OO1

A 20 MB database containing 20000 interconnected parts is used along with the queries:

`lookup:` A set of 1000 random part identifiers is generated. 10 transactions are then executed, each of which fetches the set of parts from the database.

`scan:` All parts in the database are fetched once in index order.

`traverse:` 10 transactions are executed. Each transaction selects a part at random and recursively traverses the connected parts, down to a depth of 7 (total of 3280 parts, with possible duplicates). A null procedure is called for each part traversed.

`insert`: 10 transactions are executed. Each transaction enters 100 new parts into the database and commits. Each new part is connected to 3 other (randomly selected) parts.

`insertLarge`: Generates the same workload as insert, except that 100 transactions are executed.

`update`: 500 update transactions are executed. Each transaction reads and updates 10 parts chosen at random from a contiguous range of 10% of the parts in the database. The index of the first part in each range is chosen at random for each transaction.

`lookup2`: Generates the same workload as `lookup`.

`scan2`: All parts in the database are fetched once in index order.

`traverse2`: Generates the same workload as `traverse`.

OO1b

A 50 MB database containing 40000 interconnected parts is used. A number of alterations are made to the queries of the OO1 benchmark to produce the OO1b benchmark. These are:

`lookup`: The number of random part identifiers generated is increased to 10000.

`insert`: The number of parts entered into the database by each transaction is increased to 500.

`insertLarge`: The number of parts entered into the database by each transaction is increased to 500.

`update`: The number of parts read and updated by each transaction is increased to 40.

`lookup2`: The number of random part identifiers generated is increased to 10000.

OO7

The small 20 MB OO7 database with the following configuration is used:

Parameter	Value	Parameter	Value
numAtomicPerComp	50	numAssPerAss	3
numConnPerAtomic	3	numAssLevels	7
documentSize (bytes)	2000	numCompPerAss	3
manualSize (bytes)	100K	numModules	1
numCompPerModule	500		

The queries employed are:

`T1`: The assembly hierarchy is traversed, visiting the unshared composite parts of each base assembly visited. As each composite part is visited, a depth-first traversal is carried out on its subgraph of atomic parts.

`T6`: Traversal `T1` is repeated, visiting only the root part of each composite part.

`Q2`: A range of build dates which contains the last 1% of the dates found in the database's atomic parts is chosen and these parts are retrieved.

`Q8`: All pairs of documents and atomic parts where the document identifier in the atomic part matches the identifier of the document are found.

`S2`: The 5 most recently created composite parts are removed in their entirety, including document objects and atomic part subgraphs.

MOB

A 75 MB database containing 18000 large parts (4096 bytes each) is used.

`scan`: All objects in the database are read once in index order.

`readTrans`: 1000 read-only transactions are executed. Each transaction reads 10 objects chosen at random from a contiguous range of 1800 objects (10% of the database). The index of the first object in each range is chosen at random for each transaction.

`randomAcc`: 18000 objects chosen at random are accessed.

`updateTrans`: 1000 update transactions are executed. Each transaction reads and updates 10 objects chosen using the selection algorithm used in `readTrans` and commits.

`RWtrans`: 1000 update transactions are executed. Each transaction reads 10 objects, chosen using the selection algorithm used in `readTrans`, updates the last object accessed and commits.

`randRWtrans`: 1000 update transactions are executed. Each transaction reads 10 objects chosen at random from the database, updates the last object accessed and commits.

`scan2`: Generates the same workload as `scan`.

`readTrans2`: Generates the same workload as `readTrans`.

`randomAcc2`: Generates the same workload as `randomAcc`.

Appendix B Calibrating MaStA I/O Patterns

The average cost of the I/O access patterns (*sequential*, *ordered*, *clustered*, *unclustered* and *disk*) used in MaStA are calibrated by executing synthetic I/O traces of read and write operations on raw disk partitions. The traces are recorded using the MaStA I/O trace format (Section 5.4.4). The localities of the I/O operations recorded in the traces are controlled to simulate the various access I/O patterns:

- *Sequential* I/O operations are simulated by performing I/O operations on contiguous blocks on the raw partition.
- *Clustered* I/O operations are simulated by choosing at random 10% (1920) of the blocks on a 15 MB partition and accessing the blocks in the random order.
- *Unclustered* and *disk* I/O operations are simulated by choosing at random 1920 blocks on a 150 MB partition and a 500 MB partition respectively and accessing the blocks in the random orders.
- *Ordered* I/O operations are simulated by sorting the blocks used in *unclustered* I/O traces and accessing the sorted blocks in order.

Each I/O trace is executed 5 times on a single-user system and timings are obtained using the operating systems' *time* commands. There was less < 2% variation between the 5 runs of any synthetic I/O trace.

The average I/O access patterns are given in Table 4.6 as ratios of *sequential* reads. The absolute values measured are given here as numbers of milliseconds per 8 Kbyte block for use in Chapters 6 and 7.

I/O Access Pattern	Alpha	SPARCStation
Sequential reads (r_{seq})	2.47	3.34
Sequential writes (w_{seq})	3.97	3.34
Ordered reads (r_{asc})	9.41	9.10
Ordered writes (w_{asc})	5.86	8.56
Clustered reads (r_{clu})	9.43	13.50
Clustered writes (w_{clu})	7.70	12.56
Unclustered reads (r_{uncl})	10.53	15.47
Unclustered writes (w_{uncl})	9.21	16.41
Disk reads (r_{disk})	12.21	15.38
Disk writes (w_{disk})	10.73	17.49

Appendix C Validation Results

C.1 Results of Validating I/O Assumption

In Section 6.3 the workloads of OO1, OO1b, MOB and OO7 are executed on the AISP database, the LSD and DataSafe, on the Sun and the Alpha. The average I/O costs and total costs (seconds) measured on the Sun are:

	Sun					
	AISP		DataSafe		LSD	
Workload	**Total**	**I/O**	**Total**	**I/O**	**Total**	**I/O**
lookup (OO1)	112.32	101.21	113.73	101.79	112.95	101.64
scan (OO1)	18.37	12.64	18.68	12.74	18.25	12.51
traverse (OO1)	27.55	23.21	27.86	23.29	27.36	23.02
insert (OO1)	24.84	21.74	23.92	20.13	20.90	17.76
insertLarge (OO1)	165.99	152.73	160.09	139.31	124.66	111.16
update (OO1)	170.85	160.15	148.12	131.74	119.74	108.61
lookup2 (OO1)	205.83	188.34	192.53	173.65	230.32	212.46
scan2 (OO1)	30.73	23.7	23.52	16.23	33.40	26.38
traverse2 (OO1)	28.24	24.01	26.73	22.27	31.27	27.04
lookup (OO1b)	1309.86	1206.17	1323.71	1210.71	1316.94	1207.44
scan (OO1b)	25.14	17.02	24.93	16.50	25.08	16.97
traverse (OO1b)	80.20	71.78	80.93	71.87	80.30	71.58
insert (OO1b)	85.89	77.67	95.07	84.93	76.63	68.20
insertLarge (OO1b)	784.19	721.85	870.97	790.90	698.33	633.61
update (OO1b)	688.81	648.91	717.13	661.97	590.35	548.34
lookup2 (OO1b)	3159.59	2934.80	2893.52	2649.21	3696.46	3458.29
scan2 (OO1b)	73.64	59.48	52.45	37.66	81.64	67.33
traverse2 (OO1b)	89.81	81.49	79.70	70.74	104.66	96.02
scan (MOB)	66.66	41.91	67.32	41.63	66.43	41.56
readTrans (MOB)	193.52	170.77	194.48	170.41	193.68	170.54
randomAcc (MOB)	368.70	328.91	370.23	328.00	368.78	328.16
updateTrans (MOB)	527.17	490.63	570.66	524.09	372.71	335.00
RWtrans (MOB)	371.33	342.99	324.73	290.91	317.75	288.25
randRWtrans (MOB)	372.79	344.25	335.22	301.15	323.90	294.16
scan2 (MOB)	160.53	135.80	67.37	41.65	166.12	141.12
readTrans2 (MOB)	218.84	196.09	194.75	170.68	236.84	213.67
randomAcc2 (MOB)	394.28	354.44	370.05	327.83	427.07	386.39
T1 (OO7)	61.95	50.89	62.85	51.34	62.12	50.86
T6 (OO7)	28.82	25.35	29.81	26.11	28.89	25.34
Q2 (OO7)	6.30	3.99	6.36	3.94	6.37	4.05
Q8 (OO7)	34.14	24.01	38.29	27.84	33.99	23.80
S2 (OO7)	11.41	8.27	10.26	6.99	11.34	8.19

The average I/O costs and total costs (seconds) measured on the Alpha are:

	Alpha					
	AISP		DataSafe		LSD	
Workloads	**Total**	**I/O**	**Total**	**I/O**	**Total**	**I/O**
lookup (OO1)	70.63	68.59	70.90	68.67	70.20	70.20
scan (OO1)	9.74	8.85	9.62	8.80	9.81	9.81
traverse (OO1)	17.05	16.39	17.08	16.41	16.65	16.65
insert (OO1)	15.27	14.86	15.21	14.72	13.87	13.87
insertLarge (OO1)	103.87	101.44	105.18	101.88	87.96	87.96
update (OO1)	103.08	101.27	94.11	91.69	81.50	81.50
lookup2 (OO1)	128.98	125.65	128.84	125.40	148.80	148.80
scan2 (OO1)	16.77	15.61	13.89	12.80	18.79	18.79
traverse2 (OO1)	16.86	16.18	16.24	15.57	19.19	19.19
lookup (OO1b	843.05	820.60	840.70	817.98	838.20	838.20
scan (OO1b)	13.08	11.75	12.94	11.69	13.08	14.71
traverse (OO1b)	51.91	50.40	51.66	50.04	51.33	51.33
insert (OO1b)	55.58	54.07	61.94	60.20	51.74	51.74
insertLarge (OO1b)	504.78	491.43	570.07	555.16	487.08	487.08
update (OO1b)	439.72	431.64	480.86	470.66	416.02	416.02
lookup2 (OO1b)	2036.25	1989.46	1933.64	1882.62	2513.02	2513.02
scan2 (OO1b)	41.22	38.70	34.51	32.04	53.14	53.14
traverse2 (OO1b)	56.87	55.31	51.20	49.65	70.15	70.15
scan (MOB)	26.81	20.38	26.25	19.65	26.83	26.83
readTrans (MOB)	119.16	111.35	119.00	111.31	119.25	119.25
randomAcc (MOB)	217.56	203.75	216.76	202.89	217.42	217.42
updateTrans (MOB)	312.05	294.58	352.89	331.49	261.80	261.80
RWtrans (MOB)	214.95	202.31	184.12	171.23	209.78	209.78
randRWtrans (MOB)	217.11	204.44	185.13	172.32	214.55	214.55
scan2 (MOB)	95.19	88.65	26.43	19.94	105.49	105.49
readTrans2 (MOB)	135.14	127.46	118.65	110.97	154.45	154.45
randomAcc2 (MOB)	244.22	230.60	216.74	203.08	277.34	277.34
T1 (OO7)	37.99	34.95	38.99	35.63	38.00	34.73
T6 (OO7)	18.37	17.27	19.25	18.04	18.33	17.24
Q2 (OO7)	3.55	3.19	3.37	3.01	3.44	3.09
Q8 (OO7)	21.97	19.49	26.14	23.13	22.29	19.53
S2 (OO7)	6.48	5.87	6.41	5.73	6.48	5.82

C.2 Results of Validating Cost Category Interaction Assumption

In Section 6.4 the workloads of benchmarks OO1, OO1b, MOB and OO7 are executed on the AISP database, DataSafe and the LSD, recording traces of the I/O operations performed. Each I/O trace is then ordered by MaStA I/O cost category (database reads, log writes, etc.) to produce a second set of traces. The original traces and the ordered traces are run on the raw disk partitions to measure the I/O costs. The average costs (seconds) of running the original and the ordered I/O traces on the Sun are:

	Sun					
	AISP		DataSafe		LSD	
Workloads	**Original**	**Ordered**	**Original**	**Ordered**	**Original**	**Ordered**
lookup (OO1)	99.65	98.51	99.42	98.49	99.07	98.49
scan (OO1)	13.14	13.05	13.17	13.14	13.20	13.12
traverse (OO1)	22.65	22.52	22.49	22.44	22.52	22.39
insert (OO1)	20.72	20.35	19.67	19.09	17.05	17.10
insertLarge (OO1)	144.88	142.62	135.26	130.91	106.87	108.80
update (OO1)	164.81	159.61	133.47	116.92	108.73	111.23
lookup2 (OO1)	193.21	192.97	172.72	173.09	217.12	218.16
scan2 (OO1)	24.61	24.40	16.83	16.68	26.50	26.29
traverse2 (OO1)	24.02	23.74	21.40	21.29	26.93	26.67
lookup (OO1b)	1213.92	1219.12	1212.96	1220.32	1210.07	1216.71
scan (OO1b)	19.49	19.52	19.50	19.46	19.22	19.24
traverse (OO1b)	71.21	70.64	70.81	70.64	70.71	70.20
insert (OO1b)	76.97	76.85	85.76	83.39	69.00	69.02
insertLarge (OO1b)	721.05	722.43	812.54	788.09	634.08	632.41
update (OO1b)	686.53	683.50	698.82	618.58	558.27	554.80
lookup2 (OO1b)	3200.44	3203.21	2736.09	2713.80	3493.94	3477.44
scan2 (OO1b)	66.08	65.49	41.58	41.51	68.98	68.54
traverse2 (OO1b)	86.77	86.18	69.22	68.77	95.43	95.01
scan (MOB)	38.89	39.01	38.66	38.91	38.89	38.99
readTrans (MOB)	180.09	180.23	178.10	178.54	176.23	176.79
randomAcc (MOB)	354.26	353.54	354.33	356.15	347.21	349.65
updateTrans (MOB)	517.11	507.44	570.55	483.22	342.10	338.55
RWtrans (MOB)	365.95	360.83	302.06	270.15	300.23	295.76
randRWtrans (MOB)	366.97	359.06	316.50	287.64	302.57	297.63
scan2 (MOB)	142.87	142.54	38.64	38.93	147.37	147.63
readTrans2 (MOB)	210.48	210.57	178.60	178.78	219.57	219.08
randomAcc2 (MOB)	374.36	374.84	355.25	355.27	397.28	398.32
T1 (OO7)	51.59	50.75	51.46	50.87	51.63	50.68
T6 (OO7)	26.08	25.81	26.24	25.96	26.17	25.84
Q2 (OO7)	3.90	3.88	3.86	3.87	3.92	3.94
Q8 (OO7)	23.82	23.92	27.77	27.76	23.77	23.75
S2 (OO7)	8.32	8.24	6.95	6.89	8.42	8.42

The average costs (seconds) of running the original and the ordered I/O traces on the Alpha are.

	Alpha					
	AISP		DataSafe		LSD	
Workloads	**Original**	**Ordered**	**Original**	**Ordered**	**Original**	**Ordered**
lookup (OO1)	68.97	68.76	67.33	67.59	67.58	67.60
scan (OO1)	8.62	8.69	8.54	8.57	8.64	8.70
traverse (OO1)	16.01	16.07	15.08	15.06	15.07	15.09
insert (OO1)	13.42	13.71	22.32	23.77	12.34	12.62
insertLarge (OO1)	93.78	91.05	189.47	188.61	80.46	80.50
update (OO1)	99.40	92.88	185.29	181.91	79.73	75.03
lookup2 (OO1)	126.20	126.68	125.90	126.76	146.75	146.49
scan2 (OO1)	15.86	15.88	12.30	12.26	17.83	17.80
traverse2 (OO1)	16.01	16.04	14.53	14.50	17.46	17.68
lookup (OO1b)	836.25	836.52	836.17	837.06	836.35	836.75
scan (OO1b)	12.22	12.25	12.08	12.03	12.26	12.32
traverse (OO1b)	50.84	50.86	50.91	50.83	51.54	50.80
insert (OO1b)	50.46	50.04	85.15	84.92	48.38	48.29
insertLarge (OO1b)	472.65	461.93	840.75	813.81	456.39	455.94
update (OO1b)	434.86	423.93	757.20	714.27	413.38	407.03
lookup2 (OO1b)	2030.61	2029.48	1949.89	1949.78	2513.16	2513.07
scan2 (OO1b)	41.00	40.99	33.10	33.13	48.84	48.89
traverse2 (OO1b)	55.39	55.18	50.33	49.80	69.59	69.52
scan (MOB)	26.08	26.33	25.88	25.93	25.89	25.93
readTrans (MOB)	126.81	127.02	126.39	126.57	126.42	126.08
randomAcc (MOB)	230.74	230.94	230.26	230.50	230.47	230.64
updateTrans (MOB)	313.10	300.64	639.60	576.52	262.36	252.45
RWtrans (MOB)	211.49	204.90	278.82	264.64	216.56	196.44
randRWtrans (MOB)	213.15	206.53	281.53	265.63	222.39	200.70
scan2 (MOB)	94.73	95.04	25.88	25.94	106.13	105.85
readTrans2 (MOB)	135.01	134.72	126.78	126.38	155.88	155.71
randomAcc2 (MOB)	243.39	243.50	230.41	230.66	279.87	279.67
T1 (OO7)	34.84	35.93	39.60	41.40	35.43	36.12
T6 (OO7)	16.98	18.07	20.82	20.83	17.05	18.31
Q2 (OO7)	3.02	3.07	2.96	2.96	3.11	3.15
Q8 (OO7)	18.59	20.49	35.11	35.35	18.58	20.48
S2 (OO7)	5.90	6.18	5.79	5.79	5.71	6.23

C.3 Results of Validating Access Pattern Cost Assumption

In Section 6.5 each operation recorded in the I/O traces in Section 6.4 is assigned the appropriate predicted I/O cost according to the predicted I/O access pattern performed. For example, in DataSafe, log writes are predicted to be performed sequentially and so in this procedure each log write recorded in a trace generated from DataSafe is assigned the predicted cost of a *sequential* write. The predicted costs of the I/O access patterns are taken from Appendix B. Assigning a predicted I/O cost to each operation recorded in the traces results in a predicted I/O cost for each workload running on each recovery mechanism. The predicted I/O costs and the total real costs of the workloads on the Sun are:

	Sun					
	AISP		DataSafe		LSD	
Workloads	**Total Real**	**Pred. I/O**	**Total Real**	**Pred. I/O**	**Total Real**	**Pred. I/O**
lookup (OO1)	112.32	113.24	113.73	98.87	112.95	120.13
scan (OO1)	18.37	17.19	18.68	15.04	18.25	18.22
traverse (OO1)	27.55	26.80	27.86	23.43	27.36	28.42
insert (OO1)	24.84	23.02	23.92	20.13	20.90	19.94
insertLarge (OO1)	165.99	151.42	160.09	136.80	124.66	115.22
update (OO1)	170.85	154.90	148.12	142.93	119.74	113.44
lookup2 (OO1)	205.83	201.08	192.53	175.53	230.32	213.35
scan2 (OO1)	30.73	25.68	23.52	22.45	33.40	27.23
traverse2 (OO1)	28.24	25.45	26.73	22.26	31.27	26.99
lookup (OO1b)	1309.86	1365.05	1323.71	1191.78	1316.94	1448.49
scan (OO1b)	25.14	25.68	24.93	22.45	25.08	27.23
traverse (OO1b)	80.20	84.33	80.93	73.64	80.30	89.46
insert (OO1b)	85.89	82.04	95.07	81.87	76.63	74.13
insertLarge (OO1b)	784.19	734.25	870.97	764.76	698.33	644.45
update (OO1b)	688.81	646.44	717.13	698.26	590.35	556.92
lookup2 (OO1b)	3159.59	3089.97	2893.52	2697.57	3696.46	3278.86
scan2 (OO1b)	73.64	63.33	52.45	55.29	81.64	67.19
traverse2 (OO1b)	89.81	83.02	79.70	72.50	104.66	88.07
scan (MOB)	66.66	143.55	67.32	125.21	66.43	152.30
readTrans (MOB)	193.52	203.14	194.48	177.30	193.68	215.53
randomAcc (MOB)	368.70	367.24	370.23	320.52	368.78	389.67
updateTrans (MOB)	527.17	468.94	570.66	549.89	372.71	357.96
RWtrans (MOB)	371.33	337.41	324.73	303.59	317.75	298.56
randRWtrans (MOB)	372.79	339.59	335.22	304.17	323.90	300.89
scan2 (MOB)	160.53	143.60	67.37	125.28	166.12	152.36
readTrans2 (MOB)	218.84	203.23	194.75	177.38	236.84	215.63
randomAcc2 (MOB)	394.28	367.19	370.05	320.52	427.07	389.62
T1 (OO7)	61.95	59.58	62.85	52.56	62.12	61.40
T6 (OO7)	28.82	29.62	29.81	27.18	28.89	29.74
Q2 (OO7)	6.30	5.13	6.36	4.53	6.37	5.41
Q8 (OO7)	34.14	40.05	38.29	39.15	33.99	37.08
S2 (OO7)	11.41	10.58	10.26	9.85	11.34	10.46

The predicted I/O costs and the total real costs of the workloads on the Alpha are:

	Alpha					
	AISP		DataSafe		LSD	
Workloads	**Total Real**	**Pred. I/O**	**Total Real**	**Pred. I/O**	**Total Real**	**Pred. I/O**
lookup (OO1)	70.63	77.12	70.90	69.00	70.20	81.72
scan (OO1)	9.74	11.70	9.62	10.46	9.81	12.39
traverse (OO1)	17.05	18.25	17.08	16.32	16.65	19.33
insert (OO1)	15.27	15.25	15.21	14.45	13.87	14.40
insertLarge (OO1)	103.87	98.96	105.18	97.75	87.96	87.12
update (OO1)	103.08	94.80	94.11	91.08	81.50	80.49
lookup2 (OO1)	128.98	136.94	128.84	122.54	148.80	145.14
scan2 (OO1)	16.77	17.49	13.89	15.63	18.79	18.52
traverse2 (OO1)	16.86	17.34	16.24	15.50	19.19	18.36
lookup (OO1b)	843.05	929.58	840.70	832.19	838.20	985.32
scan (OO1b)	13.08	17.49	12.94	15.63	13.08	18.52
traverse (OO1b)	51.91	57.43	51.66	51.39	51.33	60.85
insert (OO1b)	55.58	55.48	61.94	58.72	51.74	53.76
insertLarge (OO1b)	504.78	496.29	570.07	545.06	487.08	473.53
update (OO1b)	439.72	429.92	480.86	483.63	416.02	405.28
lookup2 (OO1b)	2036.25	2104.26	1933.64	1883.72	2513.02	2230.48
scan2 (OO1b)	41.22	43.12	34.51	38.57	53.14	45.69
traverse2 (OO1b)	56.87	56.54	51.20	50.58	70.15	59.91
scan (MOB)	26.81	97.75	26.25	87.39	26.83	103.59
readTrans (MOB)	119.16	138.33	119.00	123.77	119.25	146.61
randomAcc (MOB)	217.56	250.08	216.76	223.78	217.42	265.06
updateTrans (MOB)	312.05	298.17	352.89	356.42	261.80	261.35
RWtrans (MOB)	214.95	208.21	184.12	183.01	209.78	197.24
randRWtrans (MOB)	217.11	209.70	185.13	183.45	214.55	198.81
scan2 (MOB)	95.19	97.78	26.43	87.44	105.49	103.63
readTrans2 (MOB)	135.14	138.39	118.65	123.83	154.45	146.67
randomAcc2 (MOB)	244.22	250.05	216.74	223.78	277.34	265.04
T1 (OO7)	37.99	40.50	38.99	37.13	38.00	42.15
T6 (OO7)	18.37	20.18	19.25	19.38	18.33	20.72
Q2 (OO7)	3.55	3.50	3.37	3.12	3.44	3.68
Q8 (OO7)	21.97	27.30	26.14	28.72	22.29	26.82
S2 (OO7)	6.48	7.21	6.41	7.03	6.48	7.34

C.4 Results of Validating Workload Assumption

The workloads generated by OO1, OO1b, OO7 and MOB are characterised in Section 6.6 by a number of workload variables. These variables drive a synthetic workload generator that produces workloads with similar numbers of data reads and writes, and similar locality properties to the original applications. The I/O costs of executing the synthetic workloads on each recovery mechanism are measured. The average I/O costs of the synthetic database workloads and the total real costs of the original workloads on the Sun are:

	Sun					
	AISP		DataSafe		LSD	
Workloads	**Total Real**	**Synth. I/O**	**Total Real**	**Synth. I/O**	**Total Real**	**Synth. I/O**
lookup (OO1)	112.32	6.80	113.73	6.80	112.95	6.76
scan (OO1)	18.37	3.27	18.68	3.31	18.25	3.27
traverse (OO1)	27.55	5.27	27.86	5.05	27.36	5.24
insert (OO1)	24.84	19.38	23.92	17.97	20.90	14.90
insertLarge (OO1)	165.99	161.17	160.09	135.59	124.66	114.00
update (OO1)	170.85	172.48	148.12	139.91	119.74	127.38
lookup2 (OO1)	205.83	32.40	192.53	25.21	230.32	36.07
scan2 (OO1)	30.73	13.96	23.52	4.86	33.40	15.48
traverse2 (OO1)	28.24	14.61	26.73	5.59	31.27	16.47
lookup (OO1b)	1309.86	34.73	1323.71	36.08	1316.94	34.65
scan (OO1b)	25.14	4.17	24.93	4.12	25.08	4.12
traverse (OO1b)	80.20	5.31	80.93	5.22	80.30	5.27
insert (OO1b)	85.89	91.04	95.07	104.02	76.63	79.80
insertLarge (OO1b)	784.19	711.28	870.97	818.63	698.33	634.58
update (OO1b)	688.81	449.41	717.13	379.50	590.35	313.85
lookup2 (OO1b)	3159.59	882.90	2893.52	968.54	3696.46	1012.04
scan2 (OO1b)	73.64	17.81	52.45	7.46	81.64	19.59
traverse2 (OO1b)	89.81	17.11	79.70	6.60	104.66	19.14
scan (MOB)	66.66	8.27	67.32	8.15	66.43	8.14
readTrans (MOB)	193.52	91.24	194.48	91.22	193.68	90.80
randomAcc (MOB)	368.70	238.16	370.23	236.62	368.78	237.25
updateTrans (MOB)	527.17	714.68	570.66	787.22	372.71	560.50
RWtrans (MOB)	371.33	331.10	324.73	293.57	317.75	267.38
randRWtrans (MOB)	372.79	369.47	335.22	333.03	323.90	310.89
scan2 (MOB)	160.53	21.47	67.37	10.14	166.12	22.87
readTrans2 (MOB)	218.84	105.17	194.75	95.79	236.84	114.90
randomAcc2 (MOB)	394.28	242.47	370.05	234.30	427.07	266.09
T1 (OO7)	61.95	12.10	62.85	15.06	62.12	12.16
T6 (OO7)	28.82	14.19	29.81	13.20	28.89	12.96
Q2 (OO7)	6.30	3.04	6.36	2.19	6.37	3.03
Q8 (OO7)	34.14	24.30	38.29	25.16	33.99	22.63
S2 (OO7)	11.41	7.96	10.26	5.54	11.34	7.51

The average I/O costs of the synthetic database workloads and the total real costs of the original workloads on the Alpha are:

	Alpha					
	AISP		DataSafe		LSD	
Workloads	**Total Real**	**Synth. I/O**	**Total Real**	**Synth. I/O**	**Total Real**	**Synth. I/O**
lookup (OO1)	70.63	4.58	70.90	4.59	70.20	4.53
scan (OO1)	9.74	1.90	9.62	1.83	9.81	1.81
traverse (OO1)	17.05	3.54	17.08	3.36	16.65	3.50
insert (OO1)	15.27	13.91	15.21	13.76	13.87	11.87
insertLarge (OO1)	103.87	109.00	105.18	99.28	87.96	88.07
update (OO1)	103.08	110.39	94.11	95.44	81.50	93.15
lookup2 (OO1)	128.98	22.01	128.84	16.48	148.80	24.63
scan2 (OO1)	16.77	9.84	13.89	2.79	18.79	10.81
traverse2 (OO1)	16.86	10.62	16.24	3.62	19.19	11.67
lookup (OO1b)	843.05	22.24	840.70	21.10	838.20	21.75
scan (OO1b)	13.08	2.50	12.94	2.40	13.08	2.45
traverse (OO1b)	51.91	3.74	51.66	3.59	51.33	3.97
insert (OO1b)	55.58	63.23	61.94	71.52	51.74	58.03
insertLarge (OO1b)	504.78	478.23	570.07	558.64	487.08	456.29
update (OO1b)	439.72	292.14	480.86	269.26	416.02	243.77
lookup2 (OO1b)	2036.25	589.44	1933.64	551.44	2513.02	716.64
scan2 (OO1b)	41.22	11.88	34.51	4.37	53.14	13.85
traverse2 (OO1b)	56.87	11.85	51.20	4.32	70.15	13.95
scan (MOB)	26.81	5.16	26.25	5.13	26.83	5.21
readTrans (MOB)	119.16	61.34	119.00	61.57	119.25	61.56
randomAcc (MOB)	217.56	160.14	216.76	159.41	217.42	159.69
updateTrans (MOB)	312.05	453.25	352.89	516.44	261.80	407.81
RWtrans (MOB)	214.95	198.33	184.12	187.01	209.78	188.60
randRWtrans (MOB)	217.11	222.61	185.13	213.03	214.55	222.15
scan2 (MOB)	95.19	14.27	26.43	6.58	105.49	16.40
readTrans2 (MOB)	135.14	70.04	118.65	64.11	154.45	80.65
randomAcc2 (MOB)	244.22	161.02	216.74	158.38	277.34	186.73
T1 (OO7)	37.99	7.86	38.99	9.33	38.00	7.72
T6 (OO7)	18.37	10.05	19.25	9.68	18.33	9.59
Q2 (OO7)	3.55	2.47	3.37	1.45	3.44	2.47
Q8 (OO7)	21.97	18.92	26.14	19.24	22.29	18.11
S2 (OO7)	6.48	5.74	6.41	3.37	6.48	5.51

C.5 Results of Illustrating the Accuracy of MaStA

The workload variable values measured in Section 6.6 and the I/O pattern costs recorded in Appendix B are used to drive the MaStA cost models of AISP, DataSafe and the LSD developed in Chapter 4. The predicted I/O costs and the total real costs of the workloads on the Sun are:

	Sun					
	AISP		DataSafe		LSD	
Workloads	**Total Real**	**Pred. I/O**	**Total Real**	**Pred. I/O**	**Total Real**	**Pred. I/O**
lookup (OO1)	112.32	92.52	113.73	92.51	112.95	92.46
scan (OO1)	18.37	8.34	18.68	8.32	18.25	8.29
traverse (OO1)	27.55	22.93	27.86	22.92	27.36	22.88
insert (OO1)	24.84	21.01	23.92	19.73	20.90	16.91
insertLarge (OO1)	165.99	144.35	160.09	124.64	124.66	106.86
update (OO1)	170.85	143.80	148.12	126.92	119.74	107.92
lookup2 (OO1)	205.83	187.31	192.53	165.92	230.32	199.47
scan2 (OO1)	30.73	17.59	23.52	15.33	33.40	18.68
traverse2 (OO1)	28.24	25.51	26.73	22.23	31.27	27.11
lookup (OO1b)	1309.86	733.79	1323.71	733.79	1316.94	733.71
scan (OO1b)	25.14	9.70	24.93	9.67	25.08	9.65
traverse (OO1b)	80.20	71.50	80.93	71.47	80.30	71.45
insert (OO1b)	85.89	73.55	95.07	79.88	76.63	61.36
insertLarge (OO1b)	784.19	670.48	870.97	705.75	698.33	575.01
update (OO1b)	688.81	580.53	717.13	614.29	590.35	492.37
lookup2 (OO1b)	3159.59	2271.19	2893.52	2051.65	3696.46	2419.21
scan2 (OO1b)	73.64	42.90	52.45	37.37	81.64	45.63
traverse2 (OO1b)	89.81	81.17	79.70	70.62	104.66	86.40
scan (MOB)	66.66	60.85	67.32	60.78	66.43	60.81
readTrans (MOB)	193.52	96.41	194.48	96.34	193.68	96.36
randomAcc (MOB)	368.70	151.96	370.23	151.89	368.78	151.90
updateTrans (MOB)	527.17	294.97	570.66	304.45	372.71	220.81
RWtrans (MOB)	371.33	214.89	324.73	183.42	317.75	188.34
randRWtrans (MOB)	372.79	216.84	335.22	191.40	323.90	190.43
scan2 (MOB)	160.53	69.95	67.37	60.85	166.12	74.45
readTrans2 (MOB)	218.84	107.93	194.75	96.43	236.84	114.91
randomAcc2 (MOB)	394.28	159.94	370.05	151.88	427.07	170.31
T1 (OO7)	61.95	37.77	62.85	38.86	62.12	35.93
T6 (OO7)	28.82	27.93	29.81	25.74	28.89	27.99
Q2 (OO7)	6.30	4.93	6.36	4.33	6.37	5.18
Q8 (OO7)	34.14	26.25	38.29	26.98	33.99	22.36
S2 (OO7)	11.41	8.24	10.26	7.66	11.34	8.12

The predicted I/O costs and the total real costs of the workloads on the Alpha are:

	Alpha					
	AISP		DataSafe		LSD	
Workloads	**Total Real**	**Pred. I/O**	**Total Real**	**Pred. I/O**	**Total Real**	**Pred. I/O**
lookup (OO1)	70.63	64.97	70.90	64.97	70.20	64.95
scan (OO1)	9.74	5.86	9.62	5.85	9.81	5.84
traverse (OO1)	17.05	16.11	17.08	16.11	16.65	16.09
insert (OO1)	15.27	14.58	15.21	14.98	13.87	13.14
insertLarge (OO1)	103.87	98.89	105.18	96.68	87.96	90.44
update (OO1)	103.08	98.08	94.11	97.05	81.50	90.29
lookup2 (OO1)	128.98	129.48	128.84	116.30	148.80	147.51
scan2 (OO1)	16.77	12.16	13.89	10.78	18.79	13.83
traverse2 (OO1)	16.86	17.64	16.24	15.63	19.19	20.07
lookup (OO1b)	843.05	515.09	840.70	515.10	838.20	515.06
scan (OO1b)	13.08	6.81	12.94	6.80	13.08	6.80
traverse (OO1b)	51.91	50.22	51.66	50.21	51.33	50.21
insert (OO1b)	55.58	51.07	61.94	57.89	51.74	46.63
insertLarge (OO1b)	504.78	461.39	570.07	510.08	487.08	459.75
update (OO1b)	439.72	398.94	480.86	443.69	416.02	394.83
lookup2 (OO1b)	2036.25	1569.89	1933.64	1434.62	2513.02	1788.90
scan2 (OO1b)	41.22	29.67	34.51	26.25	53.14	33.77
traverse2 (OO1b)	56.87	56.12	51.20	49.62	70.15	63.92
scan (MOB)	26.81	42.76	26.25	42.70	26.83	42.75
readTrans (MOB)	119.16	67.52	119.00	67.47	119.25	67.51
randomAcc (MOB)	217.56	105.66	216.76	105.60	217.42	105.64
updateTrans (MOB)	312.05	201.23	352.89	227.01	261.80	184.99
RWtrans (MOB)	214.95	146.55	184.12	135.16	209.78	147.73
randRWtrans (MOB)	217.11	147.90	185.13	140.21	214.55	149.27
scan2 (MOB)	95.19	48.38	26.43	42.75	105.49	55.09
readTrans2 (MOB)	135.14	74.63	118.65	67.52	154.45	85.00
randomAcc2 (MOB)	244.22	110.58	216.74	105.59	277.34	125.97
T1 (OO7)	37.99	26.46	38.99	27.77	38.00	25.78
T6 (OO7)	18.37	19.29	19.25	18.49	18.33	21.14
Q2 (OO7)	3.55	3.41	3.37	3.05	3.44	3.85
Q8 (OO7)	21.97	18.08	26.14	20.29	22.29	17.94
S2 (OO7)	6.48	5.69	6.41	5.53	6.48	6.17

The predicted I/O costs obtained from MaStA configured with a uniform I/O cost are:

	Uniform		
	AISP	DataSafe	LSD
Workloads	**Pred. I/O**	**Pred. I/O**	**Pred. I/O**
lookup (OO1)	11.53	11.54	11.53
scan (OO1)	1.04	1.05	1.04
traverse (OO1)	2.87	2.88	2.87
insert (OO1)	3.06	3.60	3.06
insertLarge (OO1)	21.48	25.08	21.48
update (OO1)	21.25	24.77	21.25
lookup2 (OO1)	20.37	20.38	20.37
scan2 (OO1)	1.92	1.92	1.92
traverse2 (OO1)	2.78	2.79	2.78
lookup (OO1b)	91.18	91.19	91.18
scan (OO1b)	1.21	1.21	1.21
traverse (OO1b)	8.94	8.94	8.94
insert (OO1b)	10.28	12.60	10.28
insertLarge (OO1b)	90.01	112.15	90.01
update (OO1b)	78.65	99.91	78.65
lookup2 (OO1b)	246.88	246.89	246.88
scan2 (OO1b)	4.67	4.67	4.67
traverse2 (OO1b)	8.83	8.83	8.83
scan (MOB)	7.61	7.60	7.61
readTrans (MOB)	11.73	11.73	11.73
randomAcc (MOB)	17.40	17.40	17.40
updateTrans (MOB)	43.66	56.97	43.66
RWtrans (MOB)	28.47	30.13	28.47
randRWtrans (MOB)	28.68	30.34	28.68
scan2 (MOB)	7.62	7.61	7.62
readTrans2 (MOB)	11.74	11.74	11.74
randomAcc2 (MOB)	17.40	17.39	17.40
T1 (OO7)	4.90	5.46	4.90
T6 (OO7)	3.25	3.74	3.25
Q2 (OO7)	0.55	0.56	0.55
Q8 (OO7)	3.54	5.09	3.54
S2 (OO7)	0.98	1.15	0.98

Appendix D Scenario Code

D.1 Database Generator

The following code generates the database described in Chapter 7.

```
!link to standard library Napier88 functions
project PS() as Root onto
env :
begin
use Root with User, Library : env in
use Library with String, System, Format : env in
use String with length : proc( string -> int ) in
use Format with iformat : proc( int -> string ) in
use System with stabilise : proc() in
use User with bPlusTree, databaseEnv : env in
use bPlusTree with btreePackGen : proc[ t, r ]( int,t,r,
            proc( t, t -> bool ) -> singBtreePack[ t, r ] ) in
begin
    ! procedure which generates a database containing N customers
    let makeDB = proc( N : int )
    begin
        !create a dummy customer to populate the index
        let failValue = Customer( "",0,"",image 1 by 1 of off,0 )
        !create a new B+tree index
        let database = btreePackGen[ int, Customer ]( 4,
            -99,failValue, proc( p1, p2 : int -> bool ) ; p1 > p2 )

        for i = 1 to N do
        begin
            database( insert )( i, failValue )
        end

        !write the index to the database
        stabilise()

        !insert N dummy customers with index values 1 to N
        for i = 1 to N do
        begin
            let customerImage = image 64 by 64 of on ++ on ++ on ++
                                                on ++ on ++ on ++ on ++ on ++ on
            let customerName := "C" ++ iformat( i )

            !fill the name to be 24 characters in length
            let temp := length( customerName )
            for i = 1 to 24 - temp do
                customerName := customerName ++ "."

            let customerAddress := "This is the address for "
                ++ "customer " ++ customerName

            !fill the address to be 52 characters in length
            temp := length( customerAddress )
            for i = 1 to 52 - temp do
                customerAddress := customerAddress ++ "."

            !create the customer structure instance
            let customer = Customer( customerName, i,
                customerAddress, customerImage, i )

            database( insert )( i, customer )
        end

        !write out the database and the database's size
        in databaseEnv let database := database
        in databaseEnv let DBsize := N
        in databaseEnv let failValue = failValue
```

```
        stabilise()
    end

    makeDB( 65000 )
end
end
default : {}
```

D.2 Bank Application

The following code generates the bank's database workload described in Chapter 7.

```
let NUMTRANS = 20000                !number of transaction
let UPDATE_FREQ = 95                !percentage update transactions

!link to standard library Napier88 functions and the database
project PS() as Root onto
env :
begin
use Root with Library, User : env in
use Library with Arithmetical, System : env in
use Arithmetical with random : proc( int -> int ) in
use System with stabilise : proc() in
use User with databaseEnv : env in
use databaseEnv with database : singBtreePack[ int, Customer ] ;
                                failValue : Customer ;
                                DBsize : int in
begin
    let UPDATES_PER_TRANS = 2                   !number of customers per
                                                !update transaction
    let lastRandom := time()

    !procedure which sets the seed of the random number generator
    !used to ensure that each execution of the program obtains
    !the same sequence of random numbers
    let setSeed = proc( seed : int )
        lastRandom := seed

    !procedure which returns a random integer in the range
    ![lowR, upR]
    let randomValue = proc( lowR,upR : int -> int )
    begin
        lastRandom := random( lastRandom )
        lowR + ( lastRandom rem ( upR - lowR + 1 ) )
    end

    !procedure which accesses all the information of a given
    !customer
    let access = proc( C : Customer )
    begin
        !the next four lines ensure that all customer information
        !is read from the database
        let temp2 := C( name )(1|1)
        temp2 := C( address )(1|1)
        let img = image 10 by 10 of on++on++on++on++on++on++on++on
        copy limit C( picture ) to 1 by 1 at 1,1 onto img
    end

    setSeed( 10000 ) ;

    !execute the transactions
    for j = 1 to NUMTRANS do
    begin
        if randomValue( 1, 100 ) <= UPDATE_FREQ
        then        !execute an update transaction
        {
            !choose two customers at random
```

```
            let rand1 = randomValue( 1, DBsize )
            let rand2 = randomValue( 1, DBsize )

            !get pointers to the two customers from the index
            let customer1 = database( lookup )( rand1 )
            let customer2 = database( lookup )( rand2 )

            !read all the customer's information
            access( customer1 )
            access( customer2 )

            !update the balances of the two customers
            customer1( balance ) := customer1( balance ) + 1
            customer2( balance ) := customer2( balance ) - 1

            !commit the changes
            stabilise() ;
        }
        else    !execute a read-only transaction
        {
            !select a customer at random
            let rand1 = randomValue( 1, DBsize )

            !get a pointer the customer from the index
            let customer1 = database( lookup )( rand1 )

            access( customer1 )
        }
    end
end
end
default : {}
```

D.3 Building Society Application

The code used to generate the workload of the building society is similar to the bank's (Appendix D.2) except that 40000 transactions are executed:

```
let NUMTRANS = 40000
```

and fewer update transaction are executed.

```
let UPDATE_FREQ = 5
```

D.4 B+tree Implementation

The following code implements the B+tree index used by the benchmarks in Chapters 6, and by the database described in Chapter 7.

```
rec type Btree[t, r] is structure(entries : int ;
                                  leaf    : bool ;
                                  index   : *t ;
                                  pointers : *Index[t, r])
&
Index[t, r] is variant(btree : Btree[t, r] ;
                       record : r)

type singBtreePack[t, r] is structure(insert : proc(t, r) ;
                                      delete : proc(t) ;
                                      lookup : proc(t -> r))

project PS() as Root onto
```

```
env :
begin
use Root with User : env in
use User with bPlusTree : env in
in bPlusTree let btreePackGen := proc[t, r](n : int ; init : t ; failval : r ;
                                 gt : proc(t, t -> bool) ->
                                 singBtreePack[t, r])
begin
      type Tree is Btree[t, r]

      let createBtree = proc(-> Tree)
          Tree(0, true, vector 1 to (2 * n - 1) of init,
          vector 1 to (2 * n) of Index[t, r](record : failval))

      let root := createBtree()

      let moveRoot = proc(temp : Tree)
          if temp = root and temp(entries) = 0 do
              root := temp(pointers)(1)'btree

      let elementNumber = proc(ind : t ; node : Tree -> int)
      begin
          let i := 1
          while i <= node(entries) and gt(ind, node(index)(i)) do
                i := i + 1
          i
      end

      let containsKey = proc(ind : t ; node : Tree -> bool)
      begin
          let i := elementNumber(ind, node)
          i <= node(entries) and node(entries) > 0 and ~gt(node(index)(i), ind)
      end

      let removeIndex = proc(i : int ; node : Tree)
      begin
          if i <= node(entries) do
          begin
              for j = i to node(entries) - 1 do
              begin
                  node(index)(j) := node(index)(j + 1)
                  node(pointers)(j) := node(pointers)(j + 1)
              end
              node(pointers)(node(entries)) := node(pointers)(node(entries) + 1)
          end
          node(entries) := node(entries) - 1
          node(index)(node(entries) + 1) := init
      end

      let shuffleUp = proc(i : int ; node : Tree)
      begin
          for j = node(entries) to i by -1 do
          begin
              node(index)(j + 1) := node(index)(j)
              node(pointers)(j + 2) := node(pointers)(j + 1)
          end
          node(pointers)(i + 1) := node(pointers)(i)
      end

      let merge = proc(i : int ; left, right, node : Tree)
      begin
          if ~left(leaf) do
          begin
              left(index)(left(entries) + 1) := node(index)(i)
              left(entries) := left(entries) + 1
          end

          let cSize := left(entries)
          for j = 1 to right(entries) do
```

```
    begin
        left(index)(cSize + j) :=  right(index)(j)
        left(pointers)(cSize + j) := right(pointers)(j)
    end
    left(pointers)(cSize + right(entries) + 1) :=
        right(pointers)(right(entries) + 1)
    left(entries) := left(entries) + right(entries)

    if i < node(entries) do node(index)(i) := node(index)(i + 1)
    removeIndex(i + 1, node)
end

let moveEntryFromRight = proc(i : int ; child, rightSib, node : Tree)
begin
    let ent = child(entries)
    if child(leaf)
    then child(index)(ent + 1) := rightSib(index)(1)
    else child(index)(ent + 1) := node(index)(i)

    if child(leaf)
    then
    begin
        child(pointers)(ent + 2) := child(pointers)(ent + 1)
        child(pointers)(ent + 1) := rightSib(pointers)(1)
    end
    else child(pointers)(ent + 2) := rightSib(pointers)(1)

    child(entries) := child(entries) + 1
    node(index)(i) := rightSib(index)(1)
    removeIndex(1, rightSib)
end

let del := proc(ind : t ; node : Tree) ; {}

let deleteContains = proc(ind : t ; node : Tree)
begin
    let i = elementNumber(ind, node)
    let child = node(pointers)(i)'btree
    let rightSib = node(pointers)(i + 1)'btree

    case true of
    child(entries) > n - 1 :
    begin
        let predecessor := init
        let temp := node
        let tempChild := child
        while ~tempChild(leaf) do
        begin
            temp := tempChild
            tempChild := temp(pointers)(temp(entries) + 1)'btree
        end

        if tempChild(entries) = 1
        then
        begin
            let leftSib = temp(pointers)(temp(entries))'btree
            predecessor := leftSib(index)(leftSib(entries))
        end
        else predecessor := tempChild(index)(tempChild(entries) - 1)

        node(index)(i) := predecessor
        del(ind, child)
    end

    rightSib(entries) > n - 1 :
    begin
        moveEntryFromRight(i, child, rightSib, node)
        del(ind, child)
    end
```

```
    default :
    begin
        merge(i, child, rightSib, node)
        moveRoot(node)
        del(ind, child)
    end
end

let deleteNotContains := proc(ind : t ; node : Tree)
begin
    let i := elementNumber(ind, node)
    let child := node(pointers)(i)'btree

    if child(entries) > n - 1
    then del(ind, child)
    else     ! child node only has n - 1 entries
    begin
        let leftSib := child
        let rightSib := leftSib
        if i ~= 1 do leftSib := node(pointers)(i - 1)'btree

        if i ~= node(entries) + 1 do
            rightSib := node(pointers)(i + 1)'btree

        case true of
        i ~= 1 and leftSib(entries) > n - 1 :
        begin
            shuffleUp(1, child)
            child(index)(1) := node(index)(i - 1)

            if leftSib(leaf) then
            begin
                child(pointers)(1) := leftSib(pointers)(leftSib(entries))
                node(index)(i - 1) := leftSib(index)(leftSib(entries) - 1)
                removeIndex(leftSib(entries), leftSib)
            end
            else
            begin
                child(pointers)(1) :=
                    leftSib(pointers)(leftSib(entries) + 1)
                node(index)(i - 1) := leftSib(index)(leftSib(entries))
                leftSib(entries) := leftSib(entries) - 1
            end

            child(entries) := child(entries) + 1
            del(ind, child)
        end

        i ~= node(entries) + 1 and rightSib(entries) > n - 1 :
        begin
            moveEntryFromRight(i, child, rightSib, node)
            del(ind, child)
        end

        default :
        begin
            if i ~= 1
            then
            begin
                merge(i - 1, leftSib, child, node)
                moveRoot(node)
                del(ind, leftSib)
            end
            else
            begin
                merge(i, child, rightSib, node)
                moveRoot(node)
                del(ind, child)
```

```
            end
        end
    end
end

del := proc(ind : t ; node : Tree)
begin
    let contained = containsKey(ind, node)
    if node(leaf) then
        if contained do removeIndex(elementNumber(ind, node), node)
    else
        if contained then deleteContains(ind, node)
                     else deleteNotContains(ind, node)
end

let splitChild = proc(parent, child : Btree[t, r] ; i : int)
begin
    let newChild := createBtree()
    newChild(leaf) := child(leaf)
    newChild(entries) := n - 1

    for j = 1 to n - 1 do newChild(index)(j) := child(index)(j + n)
    for j = 1 to n do newChild(pointers)(j) := child(pointers)(j + n)

    if child(leaf) do
        child(pointers)(n + 1) := Index[t, r](btree : newChild)

    if child(leaf) then child(entries) := n
                   else child(entries) := n - 1

    for j = parent(entries) + 1 to i + 1 by -1 do
        parent(pointers)(j+1) := parent(pointers)(j)
    parent(pointers)(i + 1) := Index[t, r](btree : newChild)

    for j = parent(entries) to i by -1 do
        parent(index)(j + 1) := parent(index)(j)
    parent(index)(i) := child(index)(n)
    parent(entries) := parent(entries) + 1
end

rec let insertNonFull = proc(ind : t ; value : r ; node : Tree)
begin
    if node(leaf) then
    begin
        let i := elementNumber(ind, node)

        if containsKey(ind, node)
        then node(pointers)(i) := Index[t, r](record : value)
        else
        begin
            shuffleUp(i, node)
            node(index)(i) := ind
            node(pointers)(i) := Index[t, r](record : value)
            node(entries) := node(entries) + 1
        end
    end
    else
    begin
        let i := elementNumber(ind, node)
        let child := node(pointers)(i)'btree

        if child(entries) = 2 * n - 1 and
            ~(child(leaf)  and  containsKey(ind, child)) do
        begin
            splitChild(node, child , i)
            if gt(ind, node(index)(i)) do
                child := node(pointers)(i + 1)'btree
        end
```

```
            insertNonFull(ind, value, child)
        end
    end

    rec let search = proc(ind : t ; node : Tree -> r)
    begin
        let i = elementNumber(ind, node)
        if node(leaf) then
        begin
            let val := failval
            if containsKey(ind, node) do val := node(pointers)(i)'record
            val
        end
        else
        begin
            let child = node(pointers)(i)'btree
            search(ind, child)
        end
    end

    let insert = proc(ind : t ; value : r)
    begin
        if root(entries) = 2 * n - 1 then
        begin
            let newRoot := createBtree()
            newRoot(leaf) := false
            newRoot(entries) := 0
            newRoot(pointers)(1) := Index[t, r](btree : root)
            splitChild(newRoot, root, 1)
            root := newRoot
            insertNonFull(ind, value, root)
        end
        else insertNonFull(ind, value, root)
    end

    let lookup = proc(ind : t -> r); search(ind, root)

    let delete = proc(ind : t); del(ind, root)

    singBtreePack[t, r](insert, delete, lookup)
end
default : {}
```

References

[ABJ+92] Atkinson, M.P., Birnie, A., Jackson, N. & Philbrow, P.C. "Measuring Persistent Object Systems". In Proc. 5th International Workshop on Persistent Object Systems, San Miniato, Italy (1992). In Persistent Object Systems (Eds. A.Albano & R.Morrison). Springer-Verlag pp 63-85.

[AD85] Agrawal, R. & DeWitt, D. "Integrating Concurrency Control and Recovery Mechanisms: Design and Performance Evaluation". ACM Transactions on Database Systems, Vol. 10, No. 4, December 1985 pp 529-564.

[AS82] Aghili, H. & Severance, D. "A Practical Guide to the Design of Differential Files for Recovery of On-line Databases". ACM Transactions on Database Systems, 7,4 (1982) pp 540-565.

[BGH83] Berstein, P.A., Goodman, N. & Hadzilacos, V. "Recovery Algorithms for Database Systems". In Proc. IFIP 9th World Computer Congress, North-Holland, Amsterdam, September 1983 pp 799-807.

[BOP+89] Bretl, B., Maier, D., Otis, A., Penney, J., Schuchardt, B., Stein, J., Williams, E.H. & Williams, M.S. "The GemStone Data Management System". Object-Oriented Concepts, Databases and Applications, Addison Wesley, 1989 pp 283-308.

[BR91] Brown, A.L. & Rosenberg, J. "Persistent Object Stores: An Implementation Technique". In Dearle, Shaw, Zdonik (eds.), Implementing Persistent Object Bases, Principles and Practice, Morgan Kaufmann, 1991 pp 199-212.

[Bro89] Brown, A.L. "Persistent Object Stores". Ph.D. Thesis, University of St Andrews (1989).

[BT85] Bates, K. & TeGrotenhuis, M. "Shadowing Boosts System Reliability". Computer Designs, 1985.

[CBC+89] Connor, R.C.H., Brown, A.L., Carrick, R., Dearle, A. & Morrison, R. "The Persistent Abstract Machine". 3rd International Workshop on Persistent Object Systems, Newcastle, N.S.W., (January 1989) pp 80-95. In Persistent Object Systems (Eds. J. Rosenberg & D. Koch). Springer-Verlag pp 353-366.

[CDN93] Carey, M.J., DeWitt, D.J. & Naughton, J.F. "The OO7 Benchmark". In SIGMOD Conference on the Management of Data, Washington, DC, May 1993.

[Cha78] Challis, M.P. "Data Consistency and Integrity in a Multi-User Environment". Databases: Improving Usability and Responsiveness, Academic Press, 1978.

[CS92] Cattell, R.G.G. & Skeen, J. "Object Operations Benchmark". ACM Transactions on Database Systems 17,1 (1992) pp 1-31.

[Dav73] Davies, C.T. "Recovery Semantics for a DB/DC System". In Proc. ACM Annual Conference (1973) pp 136-141.

[Dav78] Davies, C.T. "Data Processing Spheres of Control". IBM Systems Journal, 17, 2 (1978) pp 179-198.

[DBF+94] Dearle, A., di Bona, R., Farrow, J., Henskens, F., Lindström, A., Rosenberg, J. & Vaughan, F. "Grasshopper: An Orthogonally Persistent Operating System". Computer Systems, Summer 1994 pp 289-312.

[EB84] Elhardt, K. & Bayer, R. "A Database Cache for High Performance and Fast Restart in Database Systems". ACM Transactions on Database Systems, Vol. 9, No. 4, December 1984 pp 503-525.

[EGL+76] Eswaran, K.P., Gray, J.N., Lorie, R.A. & Traiger, I.L. "The Notions of Consistency and Predicate Locks in a Database System". CACM 19,11 (1976) pp 624-633.

[FZT+92] Franklin, M.J., Zwilling, M.J., Tan, C.K., Carey, M.J. & DeWitt, D.J. "Crash Recovery in Client-Server EXODUS". In Proc. ACM SIGMOD Conference, San Diego, CA, June 1992.

[GAD+92] Gruber, O., Amsaleg, L., Daynes, L. & Valduriez, P. "Eos: An Environment for Object-Based Systems". In Proc. 25th Hawaii Conference on Systems Sciences, 1, 1 (1992) pp 757-768.

[Gar83] Garcia-Molina, H. "Using Semantic Knowledge for Transaction Processing in a Distributed Database". ACM Transactions on Database Systems 8, 2 (1983) pp 186-213.

[GMB+82] Gray, J.N., McJones, P., Blasgen, M., Lindsay, B., Lorie, R., Price, T., Putzolu, F. & Traiger, I.L. "The Recovery Manager of the System R Database Manager". ACM Computing Surveys 13, 2 (June 1982) pp 223-242.

[Gra78] Gray, J.N. "Notes on Database Operating Systems". LNCS 60, Springer-Verlag (1978) pp 393-481.

[Gra81] Gray, J.N. "The Transaction Concept: Virtues and limitations.". In Proc. 7th International Conference on Very Large Data Bases, Cannes, France (Sept. 1981) pp 144-154.

[GS87] Garcia-Molina, H. & Salem, K. "Sagas". In Proc. SIGMOD International Conference on Management of Data (1987) pp 249-259.

[Hag87] Hagmann, R.B. "Reimplementing the Cedar File System Using Logging and Group Commit". In Proc. 11th Symposium on Operating Systems Principles, 1987 pp 155-162.

[HD96] Hulse, D. & Dearle, A. "A Log-Structured Persistent Store". In Proc. 19th Australasian Computer Science Conference, Melbourne, Australia, Jan. 1996 pp 563-572.

[HHZ+92] Heiler, S., Haradhvala, S., Zdonik, S., Blaustein, B. & Rosenthal, A. "A Flexible Framework for Transaction Management in Engineering Environments". In Database Transaction Models For Advanced Applications, Elmagarmid, A.K. (ed), Morgan Kaufmann Publishers (1992) pp 88-121.

[HR83] Haerder T. & Reuter, A. "Principles of Transaction-Oriented Database Recovery". Computing Surveys, Vol. 15, No. 4, Dec. 1983 pp 287-317.

[KGC85] Kent, J., Garcia-Molina, H. & Chung, J. "An Experimental Evaluation of Crash Recovery Mechanisms". In Proc. 4th ACM Symposium on Principles of Database Systems (1985) pp 113-122.

[Kra87] Krablin, G.L. "Building Flexible Multilevel Transactions in a Distributed Persistent Environment". 2nd International Workshop on Persistent Object Systems, Appin, (August 1987) pp 213-234.

[Leu88] Leung, C.H.C. "Quantitive Analysis of Computer Systems". John Wiley & Sons Ltd. 1988.

[LLO+91] Lamb, C., Landis, G., Orenstein, J. & Weinreb, D. "The ObjectStore Database Systems". CACM 34, 10, (1991) pp 50-63. http://www.odi.com/products/os/techovrwv.html

[Lor77] Lorie, R.A., "Physical Integrity in a Large Segmented Database". ACM Transactions on Database Systems, Vol. 2, No. 1, March 1977 pp 91-104.

[MBC+89] Morrison, R., Brown, A.L., Connor, R.C.H. & Dearle, A. "The Napier88 Reference Manual". University of St Andrews Technical Report PPRR-77-89 (1989).

[MCM+94] Munro, D.S., Connor R.C.H., Morrison, R., Scheuerl, S. & Stemple, D.W. "Concurrent Shadow Paging in the Flask Architecture". 6th International Workshop on Persistent Object Systems, Tarascon, France (September 1994). In Persistent Object Systems (Eds. M.P.Atkinson, V.Benzaken & D.Maier). Springer-Verlag pp 16-42.

[MCM+95] Munro, D.S., Connor, R.C.H., Morrison, R., Moss, J.E.B. & Scheuerl, S.J.G. "Validating the MaStA I/O Cost Model for Database Crash Recovery Mechanisms". In Proc. OOPSLA'95 Workshop on Object Database Behaviour, Benchmarks and Performance, Austin Texas (October 1995).

[MHL+92] Mohan, C., Haderle, D., Lindsay, B., Pirahesh, H. & Schwarz, P. "ARIES: A Transaction Method Supporting Fine-Granularity Locking and Partial Rollbacks Using Write-Ahead Logging". ACM Transactions on Database Systems (TODS), 17 (1), 1992 pp 94-162.

[Mos81] Moss, J.E.B. "Nested Transactions: An Approach to Reliable Distributed Computing". Ph.D. Thesis, MIT (1981).

[NRZ92] Nodine, M.H., Ramaswamy, S. & Zdonik, S.B. "A Cooperative Transaction Model for Design Databases". In Database Transaction Models For Advanced Applications, Elmagarmid, A.K. (ed), Morgan Kaufmann Publishers (1992) pp 53-85.

[MS88] Moss, J.E.B. & Sinofsky, S. "Managing persistent data with Mneme: Designing a reliable shared object interface". In Dittrich, K.R. (ed.) Advances in Object-Oriented Database Systems: Second International Workshop on Object-Oriented Database Systems, LNCS 334, Springer-Verlag, 1988 pp 298-316.

[Mun93] Munro, D.S. "On the Integration of Concurrency, Distribution and Persistence". Ph.D. Thesis, University of St Andrews (1993).

[OLS85] Oki, B., Liskov, B. & Scheifler, R. "Reliable Object Storage to Support Atomic Actions". In Proc. 10th Symposium on Operating Systems Principles, 1985 pp 147-159.

[OS93] Orji, C.U. & Solworth, J.A. "Doubly Distorted Mirrors". In Proc. SIGMOD International Conference on Management of Data, Washington, D.C., (May 1993) pp 307-316.

[OS94] O'Toole, J. & Shrira, L. "Opportunistic Log: Efficient Installation Reads in a Reliable Object Server". Technical Report MIT/LCS-TM-506, March 1994. In Proc. 1st International Symposium on Operating Systems Design and Implementation, Monterey, CA (1994).

[PGK88] Patterson, D.A., Gibson, G. & Katz, R. "A Case for Redundant Arrays of Inexpensive Disks (RAID)". ACM SIGMOD, May 1988 pp 109-116.

[PS87] "The PS-algol Reference Manual fourth edition". Technical Report PPRR-12 (1987), Universities of Glasgow and St Andrews.

[Reu84] Reuter, A. "Performance Analysis of Recovery Techniques". ACM Transcations on Database Systems, Vol. 9, No. 4, December 1984 pp 526-559.

[RO91] Rosenblum, M. & Ousterhout, J.K. "The Design and Implementation of a Log-Structured File System". In Proc. 13th Symposium on Operating Systems Principles, 1991 pp 1-15.

[SCM+95a] Scheuerl, S.J.G., Connor, R.C.H., Morrison, R., Moss, J.E.B. & Munro, D.S. "The MaStA I/O Cost Model and its Validation Strategy". In Proc. Second International Workshop on Advances in Databases and Information Systems (ADBIS'95), Moscow, June 27-30 1995, Volume 1 pp 165-175.

[SCM+95b] Scheuerl, S.J.G., Connor, R.C.H., Morrison, R., Munro, D.S. & Moss, J.E.B. "The MaStA I/O Trace Format". Technical Report CS/95/4 (1995), University of St Andrews.

[SCM+96] Scheuerl, S.J.G., Connor, R.C.H., Morrison, R. & Munro, D.S. "The DataSafe Failure Recovery Mechanism in the Flask Architecture". In Proc. 19th Australasian Computer Science Conference, Melbourne, Australia, Jan. 1996 pp 573-581.

[SKW92] Singhal, V., Kakkad, S. V. & Wilson, P. R. "Texas: An Efficient, Portable Persistent Store". 5th International Workshop on Persistent Object Systems, San Miniato (Pisa), Italy (September 1992). In Persistent Object Systems (Eds. A. Albano & R. Morrison). Springer-Verlag pp 11-33.

[SM92] Stemple, D. & Morrison, R. "Specifying Flexible Concurrency Control Schemes: An abstract Operational Approach". Australian Computer Science Conference 15, Tasmania (1992) pp 873-891.

[SMK+93] Satyanarayanan, M., Mashburn, H.H., Kumar, P., Steere, D.C. & Kistler, J.J. "Lightweight Recoverable Virtual Memory". In Proc. 14th ACM Symposium on Operating System Principles, Asheville, NC, December 1993 pp 146-160.

[SO91] Solworth, J.A. & Orji, C.U. "Distorted Mirrors". ACM Parallel and Distributed Information Systems, 1991 pp 10-17.

[Sto86] Stonebraker, M. (Editor) "The Ingres Papers". Addison-Wesley, Reading, MA (1986).

[TW95] Tridgell, A. & Walsh, D. "The HiDIOS file system". In Proc. 4th Parallel Computing Workshop, London, Sept 1995. Fujitsu Laboratories Ltd.

[Vau94] Vaughan, F. "Implementation of Distributed Orthogonal Persistence Using Virtual Memory". Ph.D. Thesis, University of Adelaide (1994).

[VKD+92] Vaughan F., Koch, T., Dearle, A., Marlin, C. & Barter, C. "Casper: A Cached Architecture Supporting Persistence". Computing Systems 5, 3, (1992) pp 337-359.

[VDD+91] Velez, F., Darnis, V., DeWitt, D., Futtersack, P., Harrus, G., Maier, D. & Raoux, M. "Implementing the O2 object manager: some lessons". In Dearle, Shaw, Zdonik (eds.) Implementing Persistent Object Bases, Principles and Practices, Morgan Kaufman, 1991 pp 131-138.

[Wei86] Weikum, G. "A Theoretical Foundation of Multi-Level Concurrency Control". In Proc. ACM PODS (1986).

[WJN+95] Wilson, P.R., Johnstone, M.S., Neely, M. & Boles, D. "Dynamic Storage Allocation: A Survey and Critical Review". In Proc. 1995 Int'l Workshop on Memory Management, Kinross, Scotland, UK, Sept 27-29, 1995, Springer Verlag LNCS.